Conceitos Básicos de Linguagem de Programação C

Linguagem de Programação C

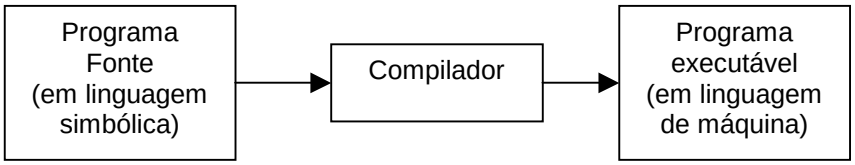
Sumário

CAPÍTULO I
Introdução

Linguagens de Programação - Definição

Um computador é uma máquina capaz de executar operações, mas para que "ele" saiba o que e como executar é preciso *programa-lo*. Um programa de computador nada mais é que um conjunto de instruções, escrito através de uma linguagem própria, que orienta o computador a realizar uma determinada tarefa.
A única linguagem que o computador "entende" é a chamada *Linguagem de Máquina*, que é formada por um conjunto de códigos numéricos, próprios para comandar a máquina. Porém, estes códigos são extremamente complicados para o entendimento humano, por isso surgiram as linguagens simbólicas.
Uma linguagem simbólica é, na realidade, um conjunto de palavras e regras facilmente compreendidas pelo homem, mas para que o computador as entenda existe a necessidade de uma tradução para a linguagem de máquina. Este processo de tradução é denominado "*compilação*".

Diagrama esquemático do processo de criação de um programa:



(a) O programa fonte é um arquivo "texto" digitado pelo programador. Este arquivo contém as instruções necessárias para que o computador realize a tarefa desejada. Utiliza-se a linguagem simbólica, genericamente chamada de *linguagem de programação*.
(b) O programa fonte é lido pelo *compilador* , que traduz as instruções da linguagem simbólica para a linguagem de máquina. Neste processo são apontados os eventuais erros existentes no programa fonte, e enquanto houver erros o compilador **não gerará** o programa executável (em linguagem de máquina), neste caso o programador deverá retornar ao editor e corrigir os erros apontados.
(c) Após o processo de compilação, se não forem detectados erros, será criado o programa executável (em linguagem de máquina). Este programa receberá o sufixo .EXE e poderá ser executado através do *prompt* do MS-DOS.

Programação em linguagem simbólica

A *linguagem C* é uma linguagem simbólica de fácil entendimento, que possui uma sintaxe bastante estruturada e flexível, tornando sua programação bem simplificada.
Para criarmos um programa, seja qual for a linguagem simbólica utilizada, devemos primeiramente definir um algoritmo capaz de resolver o problema. Na realidade o algoritmo é a estratégia que devemos utilizar para ordenar as idéias que serão, posteriormente, convertidas em instruções. Uma das ferramentas para a elaboração de algoritmos é a *pseudo-linguagem*: uma forma de escrever as instruções utilizando o português de maneira estruturada.

ATENÇÃO
Para que o estudo da linguagem C ganhe maior produtividade, esta apostila encontra-se dividida em lições que partem de um nível mais baixo até um nível mais alto de complexidade. Trabalharemos com exemplos práticos, onde, sempre que possível, mostraremos o algoritmo em linguagem Inter-S e o programa fonte correspondente em linguagem C. As instruções serão comentadas e exemplificadas no final de cada lição.

Lição 1 - Entrada/Saída

Objetivo: Criar um programa que leia 2 números do teclado e mostre o resultado da soma dos mesmos. (Nome do programa: *LICAO1*)

Solução:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição 1 // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** valor1, valor2, resultado **Numerico** **Receba** valor1 **Receba** valor2 Resultado = valor1 + valor2 **Escreva** resultado **FimRotina**	`/* Programa Licao1` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `main() {` `  int valor1, valor2, resultado;` `  scanf("%d",&valor1);` `  scanf("%d",&valor2);` `  resultado = valor1 + valor2;` `  printf("%d",resultado);` `}`

Comentários:

Observe que as primeiras linhas do algoritmo em Inter-S iniciam-se com duas barras (//). Estas linhas não são reconhecidas como instruções, servem para introduzirmos comentários quaisquer. Na codificação em linguagem C os comentários são colocados entre os caracteres /* e */. Observe que, no nosso exemplo, estas linhas trazem uma pequena documentação do programa: nome (*LICAO1*), autor (nome do programador) e data de criação, mas o programador poderá escrever o que achar necessário para documentar o programa, desde que esta documentação se inicie com os caracteres /* (barra-asterisco) e termine com os caracteres */ (asterisco-barra).

Em seguida, ainda observando a codificação do programa fonte, existem duas linhas que chamamos de *diretivas de compilação* (#include <...>). Estas diretivas serão tratadas mais adiante, no momento basta saber que elas serão necessárias para que o programa funcione adequadamente.

As linhas seguintes expressam o algoritmo propriamente dito. Na linguagem Inter-S determinamos o início do algoritmo com a palavra **Rotina**, que é expresso como **`main() {`** na codificação em linguagem C; o final da rotina é dado pela expressão **FimRotina** que na codificação em linguagem C aparece como um fechamento de chave (**`}`**).
Veja abaixo a descrição das demais linhas do algoritmo Inter-S e o formato correspondente em linguagem C:

Declare valor1, valor2, resultado **Numérico**

A expressão *Declare* processa a declaração das variáveis que serão utilizadas pela rotina. Definimos os nomes das variáveis e o tipo de dado que elas armazenarão. Neste exemplo estamos declarando as variáveis *valor1*, *valor2* e *resultado*, as quais receberão dados numéricos.
O formato desta instrução em linguagem C é o seguinte:

```
int valor1, valor2, resultado;
```

Onde `int` indica que as variáveis da lista serão numéricas inteiras (sem decimais).

Receba valor1

Esta instrução determina a leitura da entrada padrão (no nosso caso o teclado), e coloca o conteúdo lido na variável especificada: *valor1*. Em outras palavras, esta instrução aguarda que um dado seja digitado, colocando-o na variável *valor1*.
Instrução correspondente em linguagem C:

```
scanf("%d",&valor1);
```

Onde **`scanf`** é uma função para a leitura do teclado. Entre parênteses indicamos o tipo da informação a ser lida: **`"%d"`** (indica que o dado será numérico), após a vírgula informamos qual é a variável que receberá o dado: **`&valor1`**. (o símbolo & é um ponteiro necessário neste caso, que permite a alocação do valor da variável em um endereço específico de memória).

Receba valor2

Instrução idêntica à anterior, mudando apenas a variável que receberá a informação: *valor2*.
Instrução correspondente em linguagem C:

```
scanf("%d",&valor2);
```

Resultado = valor1 + valor2

Esta é uma instrução de atribuição: é efetuado um cálculo (*valor1*+*valor2*) e o resultado é colocado na variável *resultado*, ou seja, a variável *resultado* receberá a soma dos conteúdos das variáveis *valor1* e *valor2*.
Instrução correspondente em linguagem C:

```
resultado = valor1 + valor2;
```

Escreva resultado

Esta instrução envia uma informação para a saída padrão (no nosso caso o vídeo). ***Escreva*** *resultado* faz com que o conteúdo da variável *resultado* seja escrito no vídeo.
Instrução correspondente em linguagem C:

```
printf("%d",resultado);
```

Onde **`printf`** é uma função para a escrita em tela. Entre parênteses indicamos o tipo da informação a ser escrita: **`"%d"`** (indica que o dado é numérico), após a vírgula informamos qual é a variável cujo conteúdo será exibido: **`resultado`**.

Nota importante*: Toda linha de instrução em linguagem C deve ser finalizada com um ponto-e-vírgula (;)*

Digitando o programa fonte e criando o código executável

Um programa em linguagem simbólica pode ser digitado através de qualquer editor que gere textos não formatados. Podemos utilizar, por exemplo, o *EDIT* do MS-DOS ou o *Bloco de Notas* do Windows, porém, a linguagem C nos oferece um editor próprio onde, além de digitarmos o programa, podemos compila-lo gerando o programa executável. A seguir estão descritos os procedimentos para que você digite e compile o programa *LICAO1*:

- Após iniciar o Windows e efetuar o *login*, vá para o prompt do MS-DOS;

- Digite o seguinte comando: **CD\TC201** e tecle <enter> ;

- Digite **TC** e tecle <enter>. Isto fará com que o editor da linguagem C seja executado. Inicie a digitação do programa fonte. Digite o programa LICAO1 da mesma forma que ele está apresentado na página 2, mas para você não precisar virar a página, cuidamos de reproduzi-lo abaixo:

```
/* Programa Licao1
   Autor: <nome-do-aluno>
   Data.: 99/99/9999
*/
#include <stdio.h>
#include <conio.h>

main() {
  int valor1, valor2, resultado;
  scanf("%d",&valor1);
  scanf("%d",&valor2);
  resultado = valor1 + valor2;
  printf("%d",resultado);
}
```

Obs.: *Letras maiúsculas e minúsculas são interpretadas de forma diferente pelo compilador C, portanto, digite as instruções acima da mesma maneira que estão apresentadas.*

- Após a digitação grave o programa utilizando o comando *FILE /SAVE* . Será solicitado o nome com o qual o arquivo será gravado, digite LICAO1.C (todo programa fonte em linguagem C deve possuir sufixo **.C**)

- Execute o programa através do comando *RUN/RUN*. No caso de existirem erros de digitação, o programa não será executado. Neste caso o interpretador acusará as linhas erradas. Após o acerto das mesmas repete-se o processo de execução. Para criar o programa executável use o comando *COMPILE/MAKE EXE FILE*.

- Saia do editor executando o comando *FILE / EXIT*. No *prompt* do MS-DOS você poderá verificar que um arquivo com nome LICAO1.EXE foi criado (utilize o comando DIR). Este arquivo é o programa executável. Para fazê-lo funcionar basta digitar **LICAO1** e teclar <enter>.

Implementando novos comandos no programa LICAO1

Você percebeu que, do ponto de vista estético, o programa ficou "feio". A tela não foi apagada e não existem informações sobre o que fazer. O usuário quando executar este programa não saberá o que deve ser feito. Devemos, portanto, introduzir algumas instruções para melhorar o visual e tornar o programa mais "amigável". Observe como ficará o programa com a colocação dessas novas instruções:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição 1 // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** valor1, valor2, resultado **Numerico** **Limpa** **Escreva** "Programa LICAO1" **Escreva** "Digite o 1° valor:" **Receba** valor1 **Escreva** "Digite o 2° valor:" **Receba** valor2 Resultado = valor1 + valor2 **Escreva** "Soma dos valores:" **Escreva** resultado **FimRotina**	`/* Programa Licao1` `Autor: <nome-do-aluno>` `Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `main() {` `int valor1, valor2, resultado;` `clrscr();` `printf("Programa LICAO1\n");` `printf("Digite o 1° valor:");` `scanf("%d",&valor1);` `printf("Digite o 2° valor:");` `scanf("%d",&valor2);` `resultado = valor1 + valor2;` `printf("Soma dos valores:");` `printf("%d",resultado);` `}`

As novas instruções:

Limpa
Faz com que a tela seja "apagada", ou seja, deixa a tela totalmente limpa. É importante o uso dessa instrução para evitar a poluição visual da aplicação. A função correspondente em linguagem C é:

```
clrscr();
```

Escreva "Programa LICAO1"
Já vimos que a instrução **Escreva** faz com que algo seja escrito na tela. Quando a expressão a ser escrita aparece entre aspas significa que queremos que o texto seja colocado na tela da mesma forma em que foi escrito. Neste caso, estamos instruindo o computador a escrever a expressão *Programa LICAO1* no vídeo. Instrução correspondente em linguagem C:

```
printf("Programa LICAO1\n");
```

Dizemos que a expressão que aparece entre aspas é uma *constante alfanumérica*.
Observe que no final da constante foram colocados os caracteres \n (barra invertida e letra n). Em linguagem C isto significa que deve ser "saltada" uma linha após o texto escrito. Se estes caracteres forem omitidos, a próxima instrução de escrita fará a exibição do texto na mesma linha da anterior.

As outras instruções introduzidas são instruções também de escrita: **Escreva** (`printf`), que têm o mesmo formato da descrita acima.

Faça as implementações sugeridas em seu programa fonte (LICAO1.C), compile e execute-o novamente. Observe as melhorias!

Exercícios de fixação

a) Elabore um programa em linguagem C para receber três números pelo teclado e exibir a média aritmética dos mesmos. Dê-lhe o nome de *EXERC1A*. (Faça o algoritmo em Inter-S e a codificação em linguagem C). Para calcular a média utilize a seguinte fórmula:

(Valor1 + Valor2 + Valor3) / 3

Note que o operador de divisão é representado por uma barra (/). Utilize-a tanto na linguagem Inter-S como na codificação do programa em linguagem C. Os parênteses determinam a prioridade do cálculo, neste caso temos que a soma dos valores será executada antes da operação de divisão.

b) Elabore um programa para exibir o quadrado de um número digitado Dê-lhe o nome de *EXERC1B*. (Faça o algoritmo em Inter-S e a codificação em linguagem C).

Neste exercício você deverá utilizar a operação de multiplicação, cujo operador é um asterisco (*).

Lição 2 - Variáveis e Constantes

O programa proposto na lição 1 trabalhou com três variáveis: *valor1*, *valor2* e *resultado*. Percebemos, intuitivamente, que as variáveis são necessárias para que um programa receba e processe informações. Mas o que são na realidade?
Nesta lição explicaremos o que são e como utiliza-las dentro de um programa.

- **Variáveis** são posições de memória que o computador reserva para armazenar os dados manipulados pelo programa. Estas posições devem ser reservadas (declaradas) no início do programa, a partir daí ficam disponíveis para receberem qualquer conteúdo. Este conteúdo pode ser alterado quando necessário (daí o nome *variável*, pois o conteúdo pode variar).

  Na declaração de uma variável, devemos informar o nome da mesma (identificador) e qual é o tipo de dado que ela receberá.

  Os nomes ou identificadores devem obedecer às seguintes regras de construção:

  - Devem começar por uma letra (a - z , A - Z) ou um *underscore* (_).
  - O resto do identificador deve conter apenas letras, *underscores* ou dígitos (0 - 9). **Não pode** conter outros caracteres. Em C, os identificadores podem ter até 32 caracteres.
  - Letras maiúsculas são **diferentes** de letras minúsculas: Por exemplo: *MAX*, *max*, *Max* são nomes diferentes para o compilador.

  Ao "batizarmos" uma variável, ou seja, ao atribuirmos um nome à uma variável, devemos escolher um nome sugestivo, que indique o conteúdo que ela receberá. Por exemplo:
  `Valor1` → Variável que armazenará o primeiro valor
  `Valor2` → Variável que armazenará o segundo valor
  `Resultado` → Variável que armazenará o resultado de um cálculo qualquer

  É claro que o programador terá toda a liberdade de escolher o nome que quiser, mas imagine se uma variável que deva receber o valor de um produto tiver como nome **`Ze_Mane`** ? Tecnicamente não teria problema algum, pois o nome **`Ze_Mane`** está dentro das regras da linguagem, porém, no tocante à documentação do programa, teríamos um grande problema. Neste caso o programa estaria melhor documentado se o nome da variável fosse, por exemplo, **`Valor_produto`**.

  Após atribuir o nome devemos indicar o tipo de informação que a variável vai receber. Temos, basicamente, dois tipos: variáveis numéricas e variáveis alfanuméricas.

  Variáveis numéricas: São definidas para receberem dados numéricos, ou seja, números que poderão ser utilizados em cálculos diversos.
  A linguagem C classifica os dados numéricos em: *inteiro*, *ponto flutuante* e *dupla precisão*.
  Uma variável numérica inteira pode receber um valor inteiro entre -32768 a 32767.
  Uma variável numérica de ponto flutuante pode receber um número real de até 7 dígitos.
  Uma variável numérica de dupla precisão pode receber um número real de até 15 dígitos.

  Variáveis alfanuméricas: São definidas para receberem dados do tipo texto ou caractere. Podem armazenar letras, caracteres especiais, números, ou a combinação de todos eles, porém, nunca poderão ser utilizadas em cálculos, pois a característica deste tipo é meramente "texto".
  Quando definirmos uma variável alfanumérica em linguagem C, devemos indicar também o tamanho da mesma, ou seja, quantos caracteres ela poderá comportar.

- **Constantes**: Assim como as variáveis, são posições de memória que armazenam conteúdos numéricos ou alfanuméricos. As formas e regras para a construção de constantes são as mesmas das variáveis, a única diferença está no conceito de funcionamento: uma constante recebe um conteúdo inicial que não sofre alteração durante a execução do programa (é um conteúdo constante!).

O programa LICAO2, apresentado abaixo, mostra os formatos das instruções para a declaração de variáveis e constantes, tanto em Inter-S como em linguagem C:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição 2 // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** produto **Caractere** **Declare** qtde **Numerico** **Declare** preco **Numerico** **Declare** total **Numerico** **Limpa** **Escreva** "Digite o nome do produto:" **Receba** produto **Escreva** "Digite o preco do produto:" **Receba** preco **Escreva** "Digite a quantidade:" **Receba** qtde total = qtde * preco **Escreva** "O valor total do produto " **Escreva** produto **Escreva** " é: " **Escreva** total **FimRotina**	`/* Programa Licao2` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `main() {` `  char produto[10];` `  int qtde;` `  float preco;` `  double total;` `  clrscr();` `  printf("Digite o nome do produto:");` `  scanf("%10s",produto);` `  printf("Digite o preco do produto:");` `  scanf("%f",&preco);` `  printf("Digite a quantidade:");` `  scanf("%d",&qtde);` `  total = qtde * preco;` `  printf("O valor do produto ");` `  printf(produto);` `  printf(" é: ");` `  printf("%f",total);` `}`

Comentários:

O programa apresentado solicitará pelo teclado o nome do produto (que poderá ter no máximo 10 caracteres), o preço do produto (numérico real de 7 dígitos) e a quantidade (numérico inteiro). Fará o cálculo: quantidade x preço, colocando o resultado na variável total (numérico real de 15 dígitos). Finalmente exibirá no vídeo o valor total do produto.
Para uma melhor compreensão do programa, vamos analisa-lo linha a linha:

Programa fonte em Linguagem C	Comentários
`/* Programa Licao2` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `main() {` `  char produto[10];` `  int qtde;` `  float preco;` `  double total;` `  clrscr();` `  printf("Digite o nome do produto:");` `  scanf("%10s",produto);` `  printf("Digite o preco do produto:");` `  scanf("%f",&preco);` `  printf("Digite a quantidade:");` `  scanf("%d",&qtde);` `  total = qtde * preco;`	Documentação Documentação Documentação Final da documentação Diretiva de compilação Diretiva de compilação Rótulo da rotina principal Declara a variável produto como alfanumérica de 10 caracteres Declara a variável qtde como numérica inteira Declara a variável preco como numérica real de 7 dígitos Declara a variável total como numérica real de 15 dígitos Limpa a tela Escreve na tela a expressão "Digite o nome do produto" Recebe pelo teclado o conteúdo da variável produto Escreve na tela a expressão "Digite o preço do produto" Recebe pelo teclado o conteúdo da variável preco Escreve na tela a expressão "Digite a quantidade" Recebe pelo teclado o conteúdo da variável qtde Calcula qtde x preco e coloca o resultado na variável total Escreve na tela a expressão "O valor do produto "

`printf("O valor do produto ");` `printf(produto);` `printf(" é: ");` `printf("%f",total);` `}`	Escreve na tela o conteúdo da variável produto Escreve na tela a expressão " é :" Escreve na tela o conteúdo da variável total Finaliza a rotina principal

As novas instruções:

Observe que as novidades existentes no programa LICAO2 estão presentes nas declarações das variáveis e nas formas de solicitá-las pelo teclado (scanf) e escrevê-las no vídeo (printf).
No momento da declaração de uma variável devemos inicialmente informar o tipo de dado que ela receberá, utilizando um dos seguintes identificadores de tipo:

Char → Declara a variável como alfanumérica
Int → Declara a variável como numérica inteira
Float → Declara a variável como numérica real de 7 dígitos
Double → Declara a variável como numérica real de 15 dígitos

No caso da declaração de uma variável do tipo alfanumérica, devemos indicar após o nome da mesma, a quantidade de caracteres que ela suportará. Esta quantidade é expressa entre colchetes - [ ]. Na omissão desta definição, a variável suportará apenas um caracter. Exemplos:

`Char produto[10];` → Declara a variável ***produto*** como alfanumérica com o tamanho de 10 caracteres.
`Char resposta;` → Declara a variável ***resposta*** como alfanumérica com o tamanho de 1 caractere.

A função **`scanf`** (recebimento de dados pelo teclado) também deve ser ajustada de acordo com o tipo de informação. Observe a instrução abaixo:

```
scanf("%10s",produto);
```

No programa, esta instrução tem a finalidade de receber o conteúdo da variável ***produto***. Note que devemos informar o tipo e o tamanho do dado através da expressão `"%10s"`, onde o número 10 indica o tamanho e a letra s indica que a variável é uma ***String*** (o nome utilizado para identificar uma variável alfanumérica com tamanho superior a 1 caractere).

Na tabela abaixo encontramos a relação dos formatos para a declaração, recebimento pelo teclado (scanf) e escrita no vídeo (printf) dos diversos tipos de dados:

Formato da declaração	Formato do scanf	Formato do printf
`int numero;`	`scanf("%d",&numero);`	`printf("%d",numero);`
`float numero;`	`scanf("%f",&numero);`	`printf("%f",numero);`
`double numero;`	`scanf("%f",&numero);`	`printf("%f",numero);`
`char letra;`	`scanf("%c",&letra);`	`printf("%c",letra);`
`char texto[20];`	`scanf("%20s",texto);`	`printf(texto);`

Exercícios de fixação

a) Elabore um programa em linguagem C para receber as seguintes informações:

Nome : Alfanumérico de 15 caracteres;
Dia do nascimento: Numérico inteiro;
Mês do nascimento: Numérico inteiro;
Ano do nascimento: Numérico inteiro;
Valor da mesada: Numérico real de 7 dígitos.

Após o recebimento destas informações, o programa deverá apagar a tela e escrevê-las no vídeo.
Dê o nome *EXERC2A* ao programa. (Faça o algoritmo em pseudo-linguagem e a codificação em linguagem C).

b) Implemente o programa *EXERC2A* para mostrar no final a renda anual (valor da mesada multiplicado por 12). Grave-o como *EXERC2B*.

Lição 3 - Operadores e funções matemáticas

Nas lições anteriores tivemos a oportunidade de trabalhar com algumas operações matemáticas: adição, multiplicação e divisão. Nesta lição estudaremos, além dos operadores, algumas funções matemáticas importantes disponíveis na linguagem C.

Operadores

Existem cinco operadores aritméticos em C. Cada operador aritmético está relacionado a uma operação aritmética elementar: adição, subtração, multiplicação e divisão. Existe ainda um operador (**%**) chamado operador de **módulo** cujo significado é o resto da divisão inteira. Os símbolos dos operadores aritméticos são:

Operador	Operação
+	adição.
-	subtração.
*	multiplicação
/	divisão
%	módulo (resto da divisão inteira)

Exemplo: Algumas expressões aritméticas:

```
1+2   a-4.0   b*c   valor_1/taxa   num%2
```

Os operandos das expressões aritméticas devem ser constantes numéricas ou identificadores de variáveis numéricas. Os operadores **+, -, *** e **/** podem operar números de todos os tipos (inteiros ou reais) porém o operador **%** somente aceita operandos **inteiros**.

Exemplo: Expressões válidas

Expressão	Valor
6.4 + 2.1	8.5
7 - 2	5
2.0 * 2.0	4.0
6 / 3	2
10 % 3	1

Uma restrição ao operador de divisão (/) é que o denominador **deve** ser diferente de zero. Se alguma operação de divisão por zero for realizada ocorrerá um **erro de execução** do programa (*run-time error*), o programa será abortado e a mensagem *divide error* será exibida.

Quando mais de um operador se encontram em uma expressão aritmética as operações são efetuadas uma de cada vez respeitando algumas regras de precedência: Estas regras de precedência são as mesmas da matemática elementar.
Os operadores de multiplicação (*), divisão (/) e módulo (%) tem precedência sobre os operadores de adição (+) e subtração (-). Entre operadores de mesma precedência as operações são efetuadas da **esquerda** para a **direita**.

Exemplo: Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	Valor	Ordem
1 + 2 - 3	0	+ -
24 - 3 * 5	9	* -
4 - 2 * 6 / 4 + 1	2	* / - +
6 / 2 + 11 % 3 * 4	11	/ % * +

A ordem de precedência dos operadores pode ser quebrada usando-se parênteses: ( ). Os parênteses são, na verdade, operadores de mais alta precedência e são executados primeiro. Parênteses internos são executados primeiro que parênteses externos.

Exemplo: Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	Valor	Ordem
1 + (2 - 3)	0	- +
(24 - 3) * 5	105	- *
(4 - 2 * 6) / 4 + 1	-1	* - / +
6 / ((2 + 11) % 3) * 4	24	+ % / *

Observe que os operadores e os operandos deste exemplo são os mesmos do exemplo anterior. Os valores, porém, são diferentes pois a ordem de execução das operações foi modificada pelo uso dos parênteses.

O Programa LICAO3A, apresentado abaixo, demonstra algumas operações matemáticas. Ele solicita o valor do salário bruto e um percentual de desconto, calcula e exibe o valor do salário líquido:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição3A // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** bruto, liq, perc **Numerico** **Limpa** **Escreva** "Digite o salario bruto: " **Receba** bruto **Escreva** "Digite o percentual: " **Receba** perc liq = bruto-(bruto*perc/100) **Escreva** "Salario liquido: " **Escreva** liq **FimRotina**	`/* Programa Licao3A` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `main() {` `  float bruto, liq, perc;` `  clrscr();` `  printf("Digite o salario bruto:");` `  scanf("%f",&bruto);` `  printf("Digite o percentual:");` `  scanf("%f",&perc);` `  liq = bruto-(bruto*perc/100);` `  printf("Salario liquido: ");` `  printf("%f",liq);` `}`

Comentários:

Observe a instrução para o cálculo do salário líquido:

```
liq = bruto-(bruto*perc/100);
```

A utilização de parênteses é necessária neste caso. Eles determinam a prioridade do cálculo: primeiramente é efetuada a operação: *`bruto*perc/100`* (pois está entre parênteses), o resultado, em seguida, é subtraído de *`bruto`*.

Funções

Uma função é um **sub-programa** (também chamado de sub-rotina). Esta sub-rotina *recebe* informações, *as processa* e *retorna* outra informação. Por exemplo, podemos ter uma função que receba um valor numérico, calcule sua raiz quadrada e retorne o valor obtido. Existem dois tipos de funções: *funções de biblioteca* e *funções de usuário*. Funções de biblioteca são funções escritas pelos fabricantes do compilador e já estão pré-compiladas, isto é, já estão escritas em código de máquina. Funções de usuário são funções escritas pelo programador. Nesta lição trataremos somente das funções de biblioteca.

Para podermos usar uma função de biblioteca devemos **incluir** a biblioteca na compilação do programa. Esta inclusão é feita com o uso da diretiva #include colocada antes da rotina principal, como já observamos nos programas anteriores.

Vejamos algumas funções matemáticas da biblioteca **math.h**:

Funções trigonométricas do ângulo *arco*, em radianos:

```
sin(arco);  // seno
cos(arco);  // coseno
tan(arco);  // tangente
asin(arco); // arco-seno
acos(arco); // arco-coseno
atan(arco); // arco-tangente
```

Funcoes de arredondamento:

```
ceil(num) ;    // Exemplo:  ceil(3.2) == 3.0;
floor(num) ;   // Exemplo:  floor(3.2) == 4.0;
```

Funções diversas:

```
log(num);              // Logaritmo natural (base e)
log10(num);            // Logaritmo decimal (base 10)
pow(base, exp);        // Potenciacao: pow(3.2,5.6)=3.2^5.6.
sqrt(num);             // Raiz quadrada: sqrt(9.0)=3.0.
```

Para utilizar uma destas funções introduza a linha de comando **`#include <math.h>`** no cabeçalho de seu programa. A partir deste momento você poderá, dentro de qualquer rotina, utilizar qualquer função da biblioteca **math.h**. O programa abaixo exemplifica a utilização de funções matemáticas. Ele solicita um número pelo teclado, calcula e mostra a raiz quadrada e o cubo deste número:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição3B // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** num, resultado **Numerico** **Limpa** **Escreva** "Digite um número: " **Receba** num resultado = raiz(num) **Escreva** "Raiz Quadrada do número: " **Escreva** resultado resultado = num^3 **Escreva** "Cubo do número: " **Escreva** resultado **FimRotina**	`/* Programa Licao3B` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `#include <math.h>` `main() {` `  float num, resultado;` `  clrscr();` `  printf("Digite um número: ");` `  scanf("%f",&num);` `  resultado = sqrt(num);` `  printf("Raiz quadrada do número: ");` `  printf("%f",resultado);` `  resultado = pow(num,3);` `  printf("\nCubo do número: ");` `  printf("%f",resultado);` `}`

Nota: A linguagem Inter-S não possui diretivas, pois todas as funções já estão incluídas no compilador.

Exercícios de fixação

a) Elabore um programa em linguagem C que receba as variáveis numéricas de ponto flutuante (float) *w* e *z* e mostre o resultado das seguintes expressões matemáticas:

a1) $\frac{w + z}{2}$

a2) $\left(\frac{(w+100) \times \sqrt{z}}{w} \right)^3$

a3) $\sqrt{w^2 + z^2}$

Obs: Grave o programa fonte como *EXERC3*.

Lição 4 - Estruturas condicionais - Decisões *if...else*

Todos os programas vistos até agora continham instruções que eram executadas em uma seqüência linear, ou seja, as instruções eram executadas uma a uma, de cima para baixo, incondicionalmente. Uma estrutura condicional permite que o programa faça a escolha do que executar, de acordo com uma condição. A linguagem C nos oferece comandos para trabalharmos com dois tipos de estruturas condicionais: ***if...else*** e ***switch...case***. Nesta lição estudaremos a estrutura ***if...else***.

Estrutura de decisão if...else (se...então...senão)

A estrutura if...else é a mais simples estrutura de controle do C. Esta estrutura permite executar um entre vários blocos de instruções. O controle de qual bloco será executado é dado por uma *condição* (expressão lógica ou numérica). Esta estrutura pode se apresentar de dois modos diferentes: *condicional simples* ou *composta*.

Condicional simples - Decisão de um bloco (if...)

A estrutura de decisão de um bloco permite que se execute (ou não) um bloco de instruções conforme o valor de uma condição seja verdadeiro ou falso. O fluxograma desta estrutura é mostrado na figura 4.1.

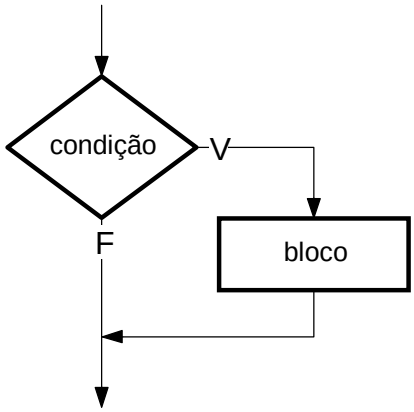


Figura 4.1: *Fluxograma da estrutura de decisão* ***if...***

O programa LICAO4A ilustra a utilização de uma estrutura condicional simples. O programa receberá pelo teclado um número, se o número for positivo será calculada e mostrada sua raiz quadrada, caso contrário, ou seja, se o número for negativo, nada será feito:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição4A // Autor : <nome-do-aluno> // Data : 99/99/9999	`/* Programa Licao4A` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `#include <math.h>`
Rotina **Declare** num, resultado **Numerico** **Limpa** **Escreva** "Digite um número: " **Receba** num **Se** num > = 0 **entao** resultado = raiz(num) **Escreva** "Raiz quadrada: " **Escreva** resultado **FimSe** **FimRotina**	`main() {` `  float num, resultado;` `  clrscr();` `  printf("Digite um número: ");` `  scanf("%f",&num);` `  if(num >= 0) {` `  resultado = sqrt(num);` `  printf("Raiz Quadrada: ");` `  printf("%f",resultado);` `  }` `  }`

A estrutura de decisão demonstrada tem como primeiro comando a avaliação de uma expressão: **Se num >= 0**, em seguida encontram-se os comandos que serão executados caso a avaliação seja verdadeira. Este bloco de comandos é finalizado com a instrução **FimSe**. Observe o uso da edentação, ou seja, o bloco de comandos encontra-se alinhado a partir de uma margem mais à direita. Este procedimento é necessário para garantir um melhor entendimento do algoritmo.
Note que na codificação em linguagem C o bloco de comandos que será executado no caso da condição ser verdadeira encontra-se entre chaves ({ }), observe também que utilizamos a edentação na codificação em linguagem C.
Reproduzimos abaixo a estrutura de decisão utilizada no programa *LICAO4A*:

```
if(num >= 0) {
    resultado = sqrt(num);
    printf("Raiz Quadrada: ");
    printf("%f",resultado);
}
```

O comando **if** (se) processa a avaliação da expressão que se encontra entre parênteses, que no nosso caso é: **num > = 0** (na realidade é feita uma pergunta: *num* é maior ou igual a zero?). Se a resposta for verdadeira o bloco de instruções que se encontra entre chaves será executado; caso a resposta seja falsa, ou seja, se *num* **não** for maior ou igual a zero, o bloco não será executado.

Condicional composta: Decisão de dois blocos (if...else)
Também é possível escrever uma estrutura que execute um entre dois blocos de instruções. A figura 4.2 mostra o fluxograma correspondente a esta estrutura de decisão.

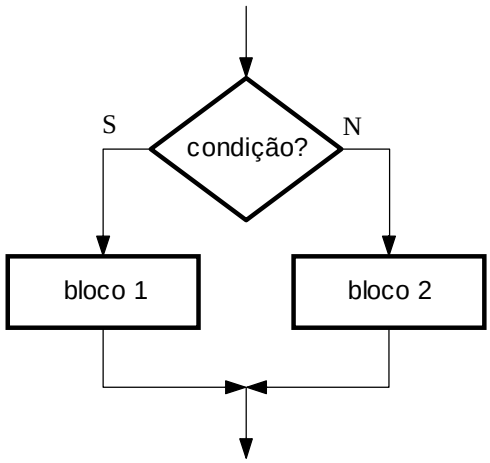


Figura 4.2: *Fluxograma da estrutura de decisão* ***if...else***

O programa LICAO4B, apresentado a seguir, ilustra a utilização de uma estrutura condicional composta. O programa receberá pelo teclado um número, se o número for par será exibida a mensagem "Número par", caso contrário será exibida a mensagem "Número ímpar":

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição4B // Autor : <nome-do-aluno> // Data : 99/99/9999	/* Programa Licao4B Autor: <nome-do-aluno> Data.: 99/99/9999 */ #include <stdio.h> #include <conio.h> #include <math.h>
Rotina **Declare** num, resto **Numerico** **Limpa**	main() { int num, resto; clrscr();

Escreva "Digite um número: " **Receba** num resto = num **mod** 2 **Se** resto = 0 **entao** **Escreva** "Número par" **Senao** **Escreva** "Número ímpar" **FimSe** **FimRotina**	`printf("Digite um número: ");` `scanf("%d",&num);` `resto = num % 2;` `if(resto == 0) {` `  printf("Número par");` `} else {` `  printf("Número ímpar");` `}` `}`

Inicialmente devemos esclarecer o processo utilizado para descobrir se o número é par ou ímpar. Sabemos que números pares são múltiplos de 2, portanto, se dividirmos um número par por 2 teremos 0 (zero) como resto da divisão. Se o número for ímpar, teremos 1 (um) como resto da divisão por 2. Observe os exemplos abaixo:

```
16 | 2
 0   8
```

16 dividido por 2 é igual a 8, com **resto 0** (zero) - indica número par.

```
15 | 2
 1   7
```

15 dividido por 2 é igual a 7, com **resto 1** (um) - indica número ímpar.

Já vimos que em linguagem C temos um operador matemático que nos retorna o resto de uma divisão: é o operador **módulo**, representado pelo caracter **%**. As divisões acima podem ser representadas em C da seguinte forma:

resto = 16 % 2 → a variável *resto* receberá conteúdo 0 (zero), pois não existe resto nesta divisão.
resto = 15 % 2 → a variável *resto* receberá conteúdo 1 (um), pois o resto da divisão é 1.

Observe no programa LICAO4B que a estrutura de decisão contém dois blocos de comandos. O primeiro encontra-se logo após o comando **`if`**, e o segundo após o comando **`else`**:

```
if(resto == 0) {
     printf("Número par");
} else {
    printf("Número ímpar");
}
```

Neste trecho do programa, primeiramente é feita a pergunta: ***resto = 0 ?***. Se a resposta for verdadeira será executado o primeiro bloco: **`printf("Número par")`** , caso contrário, ou seja, se o *resto* não for igual a zero, será executado o bloco definido logo após o comando **`else`**: **`printf("Número ímpar")`**. Observe que os dois blocos são delimitados por chaves { }.

Para concluirmos esta lição, estudaremos os operadores relacionais é lógicos, os quais nos permite a construção das expressões condicionais requeridas pelo comando if. Nos exemplos desta lição, utilizamos os operadores relacionais de *igualdade* (==) e *maior ou igual* (>=). Vamos conhecer os demais:

Operadores relacionais
Nas expressões que são avaliadas pelo comando **`if`**, utilizamos operadores de relação, os quais nos permite verificar se um determinado valor é igual ou não a um outro, se é maior ou menor, etc.
Operadores relacionais verificam a relação de magnitude e igualdade entre dois valores. São seis os operadores relacionais em C:

Operador	**Significado**
>	maior que
<	menor que
>=	maior ou igual a (não menor que)
<=	menor ou igual a (não maior que)
==	igual a
!=	não igual a (diferente de)

Operadores lógicos
Até aqui ainda não tivemos a oportunidade de utilizar os operadores lógicos. São operadores que utilizam a lógica *booleana* para a construção de expressões condicionais. Por exemplo: se eu quiser que uma determinada rotina seja executada somente quando os valores de duas variáveis (x e y) sejam iguais a zero, eu construo a seguinte expressão:

Pseudo-linguagem	Linguagem C
Se x = 0 **e** y = 0 **entao** ...	`if (x == 0 && y == 0) { ...`

Por outro lado, se eu quiser que uma determinada rotina seja executada quando o valor de uma das duas variáveis seja igual a zero, ou seja, se x for igual a zero **ou** y for igual a zero, eu construo a seguinte expressão:

Pseudo-linguagem	Linguagem C
Se x = 0 **ou** y = 0 **entao** ...	`if (x == 0 ¦¦ y == 0) { ...`

São três os operadores lógicos da linguagem C: `&&`, `¦¦` e `!`. Estes operadores têm a mesma significação dos operadores lógicos *Booleanos* AND, OR e NOT.

Exercício de fixação

Elabore um programa em linguagem C que receba um ano (numérico inteiro) e informe se o ano é bissexto ou não (anos bissextos são múltiplos de 4, portanto, se a divisão do ano por 4 gerar resto igual a zero, o ano é bissexto - use o operador **%**). Grave o programa como *EXERC4*.

Lição 5 - Estruturas condicionais - Decisões *switch...case*

A estrutura *switch...case* é uma estrutura de decisão que permite a execução de um conjunto de instruções a partir de pontos diferentes, conforme o resultado de uma expressão de controle. O resultado desta expressão é comparado ao valor de cada um dos rótulos, e as instruções são executadas a partir desde rótulo. A figura 5.1 mostra o fluxograma lógico desta estrutura.

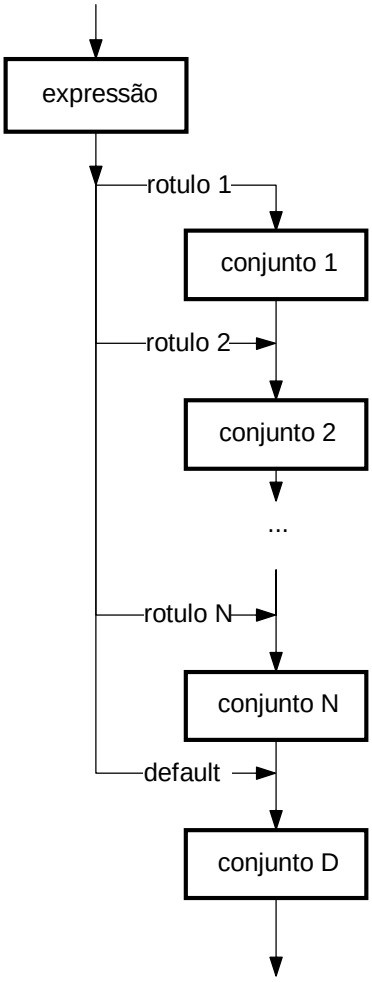


Figura 5.1: *Fluxograma da estrutura switch...case.*

O valor de *expressão* é avaliado e o fluxo lógico será desviado para o conjunto cujo *rótulo* é igual ao resultado da expressão e todas as instruções **abaixo** deste rótulo serão executadas. Caso o resultado da expressão for diferente de todos os valores dos rótulos então *conjunto d* é executado. Os rótulos devem ser expressões constantes **diferentes** entre si. O rótulo *default* é opcional.

Esta estrutura é particularmente útil quando se tem um conjunto de instruções que se deve executar em ordem, porém pode-se começar em pontos diferentes.

O programa LICAO5A, demonstrado a seguir, ilustra o uso da instrução *switch* em um menu de seleção. Neste exemplo, o programa iniciará o processo de preparação de um bolo a partir de um estágio qualquer dependendo do valor recebido:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição5A // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** selecao **Numerico** **Limpa** **Escreva** "Selecione o estágio:" **Receba** selecao **Caso** selecao **Seja =1 faca Escreva** "Obter ingredientes" **Seja =2 faca Escreva** "Juntar" **Seja =3 faca Escreva** "Bater a massa" **Seja =4 faca Escreva** "Colocar na forma" **Seja =5 faca Escreva** "Levar ao forno" **CasoNao faca Escreva** "Fim" **FimCaso** **FimRotina**	/* Programa Licao5A Autor: <nome-do-aluno> Data.: 99/99/9999 */ #include <stdio.h> #include <conio.h> main() { int selecao; clrscr(); printf("Selecione o estágio: "); scanf("%d",&selecao); switch(selecao) { case 1:printf("Obter ingredientes\n"); case 2:printf("Juntar \n"); case 3:printf("Bater a massa\n"); case 4:printf("Colocar na forma\n"); case 5:printf("Levar ao forno\n"); default:printf("Fim\n"); } }

Neste exemplo, o programa recebe um número que corresponde à seleção do estágio. Se a variável ***selecao*** receber valor 1, então serão executados os procedimentos a partir do rótulo ***case 1***; se ***selecao*** receber valor 2, serão executados os procedimentos a partir do rótulo ***case 2***, e assim por diante. Observe que existem 5 rótulos (***case 1*** a ***case 5***) e um rótulo ***default***, se a variável ***selecao*** receber um conteúdo fora do intervalo de 1 a 5, somente o procedimento após o rótulo ***default*** será executado. Lembramos que o rótulo ***default*** é opcional, ou seja, o programador pode utiliza-lo ou não.
Na codificação em Inter-S, observaremos que somente uma das instruções *SEJA* será executada, característica que difere da estrutura *switch* da linguagem C.

A instrução break.
Existem situações em uma estrutura de decisão ***switch...case*** em que desejamos que somente um procedimento seja executado (como ocorre normalmente com a instrução *CASO* no Inter-S). Para esta finalidade utilizamos a instrução ***break***, a qual faz com que a execução da estrutura seja encerrada após a execução do procedimento selecionado. O programa *LICAO5B*, demonstrado a seguir, ilustra a utilização de uma estrutura ***switch...case*** que executa apenas um dos procedimentos rotulados.

Programa fonte em Linguagem C
/* Programa Licao5B Autor: <nome-do-aluno> Data.: 99/99/9999 */ #include <stdio.h> #include <conio.h> main() { int selecao; clrscr(); printf("Selecione o estágio: "); scanf("%d",&selecao); switch(selecao) { case 1: printf("Obter ingredientes\n"); break; case 2: printf("Juntar ingredientes\n"); break;

```
     case 3: printf("Bater a massa\n"); break;
     case 4: printf("Colocar na forma\n"); break;
     case 5: printf("Levar ao forno\n"); break;
     default: printf("Fim\n");
  }
}
```

Note que o programa apresentado é parecido com o *LICAO5A*, a única diferença está na utilização da instrução ***break***. Esta instrução garante que somente um dos procedimentos será executado, de acordo com o valor da variável ***selecao***, compatibilizando o algoritmo em linguagem C com o algoritmo Inter-S

Exercício de fixação

Elabore um programa em linguagem C que receba duas variáveis numéricas e através de uma estrutura de decisão ***switch...case*** mostre a soma destas variáveis, ou a subtração, ou a multiplicação ou a divisão, de acordo com o valor recebido em uma terceira variável. Grave o programa como *EXERC5*.

Lição 6 - Estruturas de repetição

Estruturas de repetição permitem que um bloco de instruções seja executado, repetidamente, uma quantidade controlada de vezes. A linguagem C possui 3 estruturas de repetição: ***do...while***, ***while*** e ***for***.

Estrutura *do...while*

Esta é uma estrutura básica de repetição condicional. Permite a execução de um bloco de instruções repetidamente. Sua sintaxe é a seguinte:

```
do{
  bloco
}while(condição);
```

onde: *condição* é uma expressão lógica ou numérica.
bloco é um conjunto de instruções.

Esta estrutura faz com que o bloco de instruções seja executado pelo menos uma vez. Após a execução do bloco, a condição é avaliada. Se a condição é **verdadeira** o bloco é executado outra vez, caso contrário a repetição é terminada. O fluxograma desta estrutura é mostrado na figura 6.1:

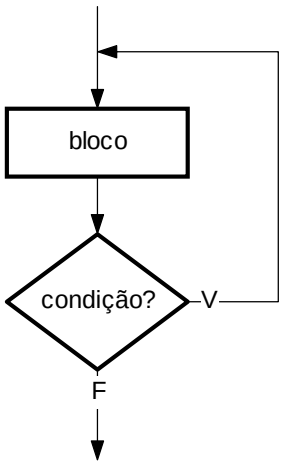


Figura 6.1: *Fluxograma da estrutura* ***do...while****.*

O programa LICAO6A, demonstrado abaixo, utiliza uma estrutura ***do...while*** para receber um número pelo teclado e mostrar o quadrado do mesmo. A estrutura será encerrada quando o número digitado for menor ou igual a zero, ou seja, será executada enquanto o número for maior que zero:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição6A // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** num **Numerico**	/* Programa Licao6A Autor: <nome-do-aluno> Data.: 99/99/9999 */ #include <stdio.h> #include <conio.h> main() { float num;

Limpa **Repita** **Escreva** "Digite um número: " **Receba** num **Escreva** "Quadrado: " **Escreva** num*num **AteQue** num <= 0 **FimRotina**	clrscr(); do { printf("\nDigite um número: "); scanf("%f",&num); printf("Quadrado: "); printf("%f", num*num); } while(num > 0); }

Observe que as instruções existentes entre os comandos ***do*** e ***while*** serão executadas enquanto o valor da variável *num* for maior que zero.
Note que no algoritmo Inter-S o final da estrutura é representado pela expressão **AteQue**, ou seja, a estrutura será repetida *até que* algo seja verdadeiro. Já no algoritmo em linguagem C a estrutura é finalizada pela expressão **while**, ou seja, a estrutura será repetida *enquanto* algo for verdadeiro.

Estrutura *while*

A estrutura de repetição condicional ***while*** é semelhante a estrutura ***do...while***. Sua sintaxe é a seguinte:

```
while(condição){
  bloco
}
```

onde: *condição* é uma expressão lógica ou numérica.
bloco é um conjunto de instruções.

Esta estrutura faz com que a condição seja avaliada em primeiro lugar. Se a condição é **verdadeira** o bloco é executado uma vez e a condição é avaliada novamente. Caso a condição seja **falsa** a repetição é terminada sem a execução do bloco. Observe que nesta estrutura, ao contrário da estrutura ***do...while***, o bloco de instruções pode não ser executado nenhuma vez, basta que a condição seja inicialmente falsa. O fluxograma desta estrutura é mostrada na figura 6.2:

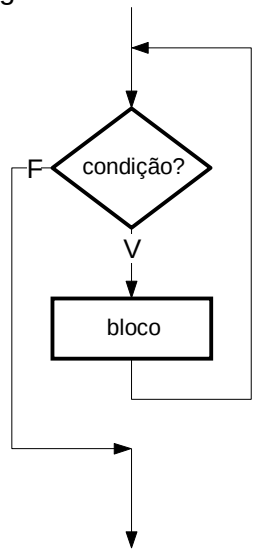


Figura 6.2: *Fluxograma da estrutura while.*

O programa *LICAO6B*, demonstrado abaixo, ilustra a utilização de uma estrutura ***while***. O programa faz o mesmo que o programa anterior (*LICAO6A*), porém, a estrutura de repetição é diferente:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição6B // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** num **Numerico** Num = 1 **Limpa** **Enquanto** num > 0 **Faca** **Escreva** "Digite um número: " **Receba** num **Escreva** "Quadrado: " **Escreva** num*num **FimEnquanto** **FimRotina**	`/* Programa Licao6B` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `main() {` `  float num;` `  num = 1;` `  clrscr();` `  while(num > 0) {` `  printf("\nDigite um numero: ");` `  scanf("%f",&num);` `  printf("Quadrado: ");` `  printf("%f", num*num);` `  }` `}`

Observe que o programa acima tem o mesmo objetivo do programa LICAO6A: recebe um número pelo teclado e mostra o quadrado do mesmo enquanto o número for maior que zero. Note também que a variável *num* foi iniciada com o valor 1 (`num = 1`), isto foi necessário para que a estrutura pudesse ser iniciada, pois se o valor de *num*, na primeira vez, fosse menor ou igual a zero, a estrutura não seria executada e o programa terminaria.

Estrutura *for*

A estrutura ***for*** é muito semelhante às estruturas de repetição vistas anteriormente, entretanto costuma ser utilizada quando se quer um número determinado de ciclos. A contagem dos ciclos é feita por uma variável chamada de **contador**. A estrutura ***for*** é, as vezes, chamada de estrutura de **repetição com contador**. Sua sintaxe é a seguinte:

```
for(inicialização; condição; incremento){
  bloco
}
```

onde: *inicialização* é uma expressão de inicialização do contador.
condição é uma expressão lógica de controle de repetição.
incremento é uma expressão de incremento do contador.
bloco é um conjunto de instruções a ser executado.

Esta estrutura executa um número determinado de repetições usando um contador de iterações. O contador é inicializado na expressão de *inicialização* **antes** da primeira iteração. Por exemplo: `i = 0`; ou `cont = 20`. Então o bloco é executado e **depois** de cada iteração, o contador é incrementado de acordo com a expressão de *incremento*. Por exemplo: `i=i+1` ou `cont = cont-2`. Então a expressão de condição é avaliada: se a condição for verdadeira, o *bloco* é executado novamente e o ciclo recomeça, se a condição for falsa termina-se o laço. Esta condição é, em geral, uma expressão lógica que determina o ultimo valor do contador. Por exemplo: `i <= 100` ou `cont > 0`.

O programa *LICAO6C*, demonstrado abaixo, ilustra a utilização da estrutura ***for***. O contador `i` é inicializado com o valor 1. O bloco é repetido enquanto a condição `i <= 10` for verdadeira. O contador é incrementado com a instrução `i=i+1`. Esta estrutura, deste modo, mostra na tela os números 1, 2, ..., 9, 10.

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição6C // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** i **Numerico** **Limpa** **Para** i=1 **Ate** 10 **Faca** **Escreva** i **FimPara** **FimRotina**	`/* Programa Licao6C` `  Autor: <nome-do-aluno>` `  Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>` `main() {` `  int i;` `  clrscr();` `  for(i=1; i<=10; i=i+1) {` `  printf("%d",i);` `  }` `}`

É interessante notar que a mesma estrutura lógica pode ser implementada usando as estruturas ***for*** ou ***do...while***:

Exemplo: As seguintes instruções são plenamente equivalentes:

```
i = 1;
do{
  printf("%d",i);
  i=i+1;
}while(i <= 10);
```

```
for(i = 1; i <= 10; i=i+1){
    printf("%d",i);
 }
```

Obs.: *Quando necessário, podemos interromper a execução dos comandos de uma estrutura de repetição utilizando a instrução* ***break****. Como já vimos na lição anterior, a instrução* ***break*** *promove o encerramento da estrutura, independentemente da condição de controle ser verdadeira ou falsa.*

Exercícios de fixação

a) Elabore um programa em linguagem C que receba um número inteiro e mostre a tabuada de 1 a 10 deste número. Utilize a estrutura ***do...while***. Grave o programa como *EXERC6A*.

b) Faça outro programa de tabuada, semelhante ao *EXERC6A*, usando agora a estrutura ***while***. Grave-o como *EXERC6B*.

c) Faça mais um programa, semelhante aos anteriores, utilizando a estrutura ***for***. Grave-o como *EXERC6C*.

Lição 7 - Desvios

Desvios são instruções que desviam a seqüência de execução dos comandos para um outro ponto especificado do programa. Temos na linguagem C, basicamente, 2 tipos de desvios incondicionais: ***goto*** (o mesmo que a instrução VaPara do Inter-S) e ***função definida pelo usuário*** (correspondente à instrução SubRotina do Inter-S).

Desvio *VaPara* (*goto*).

Esta instrução é chamada de desvio de fluxo. A instrução desvia o programa para um rótulo (posição identificada) no programa. São raros os casos onde a instrução ***goto*** é necessária, no entanto, há certas circunstâncias, onde usada com prudência, ela pode ser útil.

Sintaxe: A sintaxe da instrução ***goto*** é a seguinte:

```
goto rótulo;
...
rótulo:
...
```

onde *rótulo* é um identificador válido.

O programa LICAO7A, demonstrado abaixo, mostra um laço de repetição onde se lê valores para o cálculo de uma média. Foram usadas duas instruções ***goto***:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição7A // Autor : <nome-do-aluno> // Data : 99/99/9999 **Rotina** **Declare** val, soma **Numerico** **Declare** num **Numérico** **Limpa** **Escreva** "Digite valores:" **[Inicio]** **Escreva** "Valor: " **Receba** val **Se** val <= 0 **Entao** **VaPara** Fim **FimSe** num = num + 1 soma = soma + val **VaPara** Inicio **[Fim]** **Escreva** "Média: ", soma / num **FimRotina**	/* Programa Licao7A Autor: <nome-do-aluno> Data.: 99/99/9999 */ #include <stdio.h> #include <conio.h> main() { float val, soma; int num; clrscr(); printf("Digite valores:"); inicio: printf("\nValor:"); scanf("%f",&val); if(val <= 0){ goto fim; } num = num + 1; soma = soma + val; goto inicio; fim: printf("Média: %f",soma/num); }

Observe a existência de dois rótulos no programa acima: ***Inicio*** e ***Fim*** (note que os rótulos devem ser seguidos por *dois-pontos* :). Para processar um desvio basta utilizar a instrução ***goto*** indicando o rótulo do ponto para onde queremos desviar o fluxo.

Desvio *Função*

Na página 12 (lição 3) vimos o conceito de *funções*. Dissemos que funções são sub-programas escritos em um local fora da rotina principal (***main***).
Vimos também que existem dois tipos de funções: *de biblioteca* e *definidas pelo programador*. Na lição 3 estudamos algumas funções de biblioteca, mais precisamente da biblioteca matemática ***math.h***. Agora estudaremos as funções *definidas pelo programador*, que também podem ser chamadas de ***SubRotinas***, pois são blocos de instruções definidos em um ponto fora da rotina principal.
Diferente do que ocorre com um desvio do tipo ***VaPara***, um desvio ***SubRotina*** sempre retorna ao ponto de chamada após a execução das instruções definidas naquele procedimento.

O programa *LICAO7B*, demonstrado abaixo, ilustra a utilização de duas funções. A primeira, ***preparar_tela***, é um procedimento que não requer parâmetros, ou seja, não necessita de nenhuma informação para que as instruções nele existentes sejam executadas. A segunda, ***media***, é um procedimento que requer dois parâmetros para que seja executado. Observe o programa e em seguida faremos os devidos comentários:

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa: Lição7B // Autor : <nome-do-aluno> // Data : 99/99/9999	`/* Programa Licao7B` `Autor: <nome-do-aluno>` `Data.: 99/99/9999` `*/` `#include <stdio.h>` `#include <conio.h>`
Rotina **Declare** valor1, valor2, y **Numerico** **//** **Declare** media **Numerico** **Processe Preparar_tela** **Escreva** "Digite o 1º valor:" **Receba** valor1 **Escreva** "Digite o 2º valor:" **Receba** valor2 **Processe media** **Escreva** "Media: " **Escreva** y **FimRotina**	`main() {` `float valor1, valor2, y;` `void preparar_tela();` `float media(float, float);` `preparar_tela();` `printf("Digite o 1º valor:");` `scanf("%f",&valor1);` `printf("Digite o 2º valor:");` `scanf("%f",&valor2);` `y = media(valor1, valor2);` `printf("Média: ");` `printf("%f",y);` `}`
SubRotina Preparar_tela **Limpa** **Escreva** "Programa LICAO7B" **FimSubRotina**	`void preparar_tela() {` `clrscr();` `printf("Programa LICAO7B\n");` `}`
SubRotina Media **//** y = (a + b) / 2 **//** **FimSubRotina**	`float media(float a, float b) {` `float c;` `c = (a + b) / 2;` `return (c);` `}`

Comentários

No programa acima utilizamos 2 funções: ***preparar_tela*** e ***media***. Observe que devemos declarar estas funções assim como declaramos as variáveis, com a diferença de que devemos indicar o tipo de dado que a função retornará e o tipo dos dados que servirão de parâmetros:

Observe a instrução abaixo:

```
void preparar_tela();
```

Esta instrução está declarando a função ***preparar_tela***, indicando que ela não retornará nenhum dado, ou seja, é uma função *vazia* (void). Ao mesmo tempo informa que a função não receberá nenhum parâmetro, pois não existe nenhuma indicação entre os parênteses.

A instrução de declaração da função ***media*** tem características diferentes:

```
float media(float, float);
```

O indicador de tipo ***float***, utilizado no início da instrução, informa que a função retornará um dado numérico de ponto flutuante. Os indicadores ***float*** que se encontram entre parênteses, indicam que a função receberá dois parâmetros do tipo numérico de ponto flutuante.

Após a declaração das funções, estas podem ser chamadas em qualquer ponto dentro da rotina principal (***main***). Observe a instrução de chamada da função ***preparar_tela***:

```
preparar_tela();
```

Esta instrução promoverá a execução dos comandos existentes dentro da função ***preparar_tela***, os quais estão definidos após o encerramento da rotina principal. Após a execução destes comandos, o programa retornará à instrução subseqüente à chamada da função.

Observe agora a instrução de chamada da função ***media***:

```
y = media(valor1, valor2);
```

Esta instrução está atribuindo à variável ***y*** o valor retornado pela função ***media***, a qual terá como parâmetros os conteúdos das variáveis ***valor1*** e ***valor2***.
Note que, dentro da função, estes parâmetros serão tratados como ***a*** e ***b***. Uma outra variável está sendo declarada dentro da função: a variável ***c***. Esta variável receberá o resultado do cálculo da média de ***a*** e ***b***, o qual será o retorno da função:

```
float media(float a, float b) {
  float c;
  c = (a + b) / 2;
  return (c);
}
```

Exercícios de fixação

a) Crie o programa *EXERC7A* utilizando como base o programa *LICAO6C* (página 25), substituindo a estrutura de repetição ***for*** por desvios ***goto***.

b) Elabore um programa para calcular e mostrar a média de 4 notas bimestrais, informadas pelo teclado. Crie uma função para o cálculo. Grave o programa como *EXERC7B*.

CAPÍTULO II
Trabalhando com tela em modo texto

Comandos para recebimento e exibição de dados em tela padrão (modo texto)
Neste capítulo estudaremos os comandos básicos para o recebimento e saída de dados utilizando tela em modo texto.

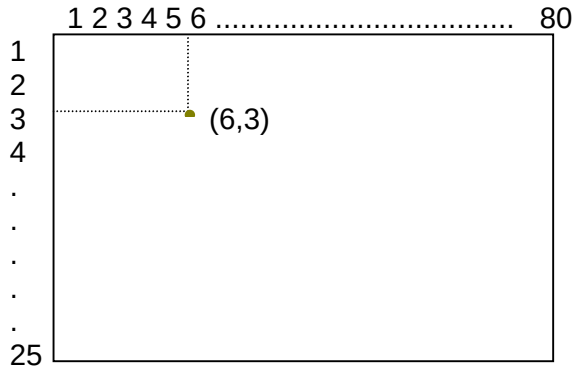
Como já sabido, o objetivo de todo e qualquer programa de computador é o processamento dos dados de entrada de forma a gerar os dados de saída. Como podemos perceber, os comandos de entrada e saída de informações sempre estarão presentes em nossos programas.
Nas outras apostilas deste site tivemos a oportunidade de utilizar as funções *scanf* (entrada de dados) e *printf* (saída de dados). Nesta apostila estudaremos estas e outras funções para o recebimento e exibição de dados, porém, antes de nos aprofundarmos neste estudo, é necessário um conhecimento sobre as funções de vídeo (tela).

A linguagem C oferece dois modos para a exibição de dados em vídeo: *modo texto* e *modo gráfico*. No modo texto podemos utilizar apenas os caracteres pertencentes à tabela ASCII* (letras, dígitos e caracteres especiais), ao passo que no modo gráfico podemos construir imagens mais sofisticadas desenhando-as ponto-a-ponto ou utilizando funções prontas para o traçado de linhas, círculos, retângulos, etc.

> ** Tabela ASCII (Americam Standard Code for Information Interchange - Tabela de código padrão americano para a troca de informações) é uma tabela adotada pelos fabricantes de computadores que contém os códigos internos para a representação dos caracteres (letras, dígitos, sinais gráficos, caracteres especiais e de controle).*

Tela em modo texto

O modo texto é o modo de tela chamado "default", ou seja, é o modo padrão.
Trabalhar no modo texto é muito simples, como você pôde perceber: basta utilizar a função *printf* para escrever as informações na tela, porém, esta função não nos permite determinar em qual local da tela a informação deve ser escrita.
A tela em modo texto contém 25 linhas (numeradas de 1 a 25) sendo que cada linha contém 80 colunas (numeradas de 1 a 80), formando assim uma matriz de 80 colunas por 25 linhas.
Uma posição de tela é referenciada pelo número da coluna seguido do número da linha. Por exemplo, a referência (6,3) representa a posição dada pela interseção da coluna 6 com a linha 3:



Em analogia ao par cartesiano, as colunas representam o eixo X e as linhas o eixo Y. Um ponto de tela (posição) é sempre referenciado pelo número da coluna seguido pelo número da linha separados por vírgula, assim como na representação cartesiana: X,Y.
A linguagem C oferece algumas funções úteis para a escrita em tela, a saber:

Antes da apresentação das funções, convém explicarmos as abreviaturas e convenções que utilizamos na representação de suas formas sintáticas (sintaxe):

const_char → Significa uma constante alfanumérica qualquer (texto qualquer entre aspas).
lista_var → Significa um nome de variável ou uma lista de variáveis separadas por vírgula.
Colchetes [ ] → Os argumentos entre colchetes são opcionais, ou seja, não são obrigatórios.

Função *printf() - (print formated) -* escrita formatada

Objetivo: Permite a escrita de informações na tela, a partir da posição do cursor.
Sintaxe.: `printf(const_char,[lista_var])`

Exemplos:

Algoritmo Inter-S	Linguagem C
Escreva "Ola pessoal"	`printf("Ola pessoal");`
Escreva vl	`printf("%d",vl);`
Escreva "Valor=" , vl	`printf("Valor= %d",vl);`

Note que nos dois últimos exemplos a variável vl é uma variável numérica, sendo assim, dentro da constante alfanumérica const_char , devemos indicar o tipo de dado que a variável armazena, através dos caracteres %d. Estes caracteres são chamados de controle, pois indicam ao compilador qual é o formato do dado que será escrito na tela. Por enquanto nossos exemplos utilizaram apenas variáveis numéricas inteiras (números inteiros) que têm o formato %d. Mais adiante estudaremos os demais tipos de variáveis suportados pela linguagem C.

Função *cprintf() - (color print formated) -* escrita formatada em cores

Objetivo: Permite a escrita de informações na tela, a partir da posição do cursor, usando cores.
Sintaxe.: `cprintf(const_char,[lista_var])`

Exemplos:

Algoritmo Inter-S	Linguagem C
Escreva "Ola pessoal"	`cprintf("Ola pessoal");`
Escreva vl	`cprintf("%d",vl);`
Escreva "Valor=" , vl	`cprintf("Valor= %d",vl);`

Note que a função *cprintf* é idêntica à função *printf*, a diferença é que *cprintf* aceita definições de cores, mas para que a saída seja colorida é necessário definir as cores de fundo e de letra antes de seu uso, isto é feito através das funções textbackground() e textcolor(), conforme a seguinte sintaxe:

```
textcolor(cor_de_letra);
textbackground(cor_de_fundo);
```

Onde *cor_de_letra* e *cor_de_fundo* são números inteiros referentes às cores da palheta padrão (16 cores no modo texto). Cada cor é representada por um número, conforme tabela abaixo:

Cor	Nº	Cor	Nº	Cor	Nº	Cor	Nº
Preto	0	Vermelho	4	Cinza escuro	8	Vermelho claro	12
Azul	1	Magenta	5	Azul claro	9	Magenta claro	13
Verde	2	Marrom	6	Verde claro	10	Amarelo	14
Ciano	3	Cinza claro	7	Ciano claro	11	Branco	15

- As cores a partir da número 8, inclusive, não podem ser utilizadas como cor de fundo.
- Para que a cor fique piscante, adicione 128 ao seu número. Veja exemplos:

Exemplo 1:

```
textcolor(15);
textbackground(1);
cprintf(" Vichinsky ");
```

Escreve a expressão Vichinsky com letras brancas (cor nº 15) sobre fundo azul (cor nº 1)

Exemplo 2:

```
textcolor(142);
textbackground(4);
cprintf(" Alerta! ");
```

Escreve a expressão Alerta! com letras amarelas piscantes (cor nº 14 + 128) sobre fundo vermelho (cor nº 4)

Função *gotoxy() - (go to position x, y) -* posicionamento do cursor

Objetivo: Colocar o cursor em uma posição definida da tela.
Sintaxe.: `gotoxy(coluna, linha)`

Esta função posiciona o cursor em uma determinada coluna (x) e linha (y) permitindo que as informações de saída sejam escritas a partir desta posição. As funções *printf* e *cprintf*, como já vimos, escrevem as informações a partir da posição do cursor, portanto, trabalhando com *gotoxy* juntamente com as funções de escrita podemos obter bons resultados de tela. Veja exemplo:

```
textcolor(15);
textbackground(1);
gotoxy(1,1); cprint(".--------------------.");
gotoxy(1,2); cprint("|  Exemplo de tela  |");
gotoxy(1,3); cprint("|    Linguagem C    |");
gotoxy(1,4); cprint("'------------------'");
```

Com este trecho de programa teremos uma moldura abrangendo as linhas 1 a 4, a partir da coluna 1, com letras brancas e fundo azul.

Função *clrscr() - (clear screen) -* limpeza de tela

Objetivo: Limpar (apagar) a tela.
Sintaxe.: `clrscr()`

Esta função faz com que a tela seja apagada, preenchendo-a com a cor definida para o segundo plano (fundo). Exemplo:

```
textbackground(2);
clrscr();
```

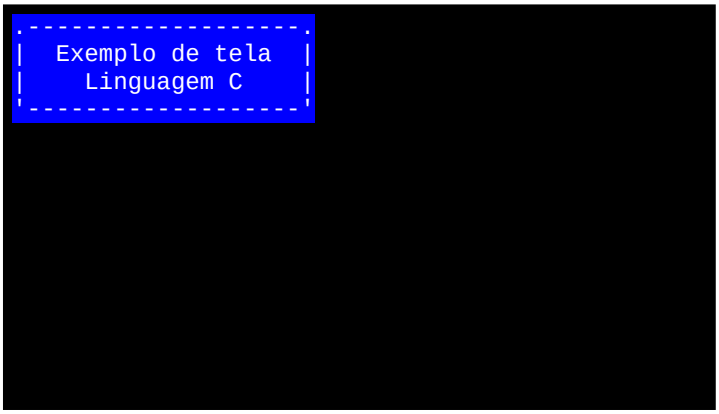
Estas instruções fazem com que a tela seja apagada, preenchendo-a com a cor nº 2 (verde).

Alguns recursos interessantes

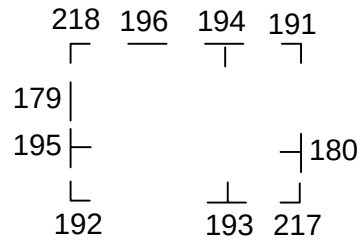
No trecho de programa que serviu como exemplo da função gotoxy, fizemos uma moldura de tela utilizando caracteres básicos do teclado. Reveja o código:

```
textcolor(15);
textbackground(1);
gotoxy(1,1); cprint(".--------------------.");
gotoxy(1,2); cprint("|  Exemplo de tela  |");
gotoxy(1,3); cprint("|    Linguagem C    |");
gotoxy(1,4); cprint("'--------------------'");
```

Estas instruções gerarão o seguinte efeito na tela:



Para dar uma estética melhor à moldura podemos utilizar o conjunto de caracteres semi-gráficos do computador. Estes caracteres não se encontram no teclado, mas podem ser extraídos através da combinação de um código numérico com a tecla ALT. Observe que para a construção da moldura utilizamos os seguintes caracteres: ponto (.), hífen (-), barra vertical (|) e apóstrofo ('), mas poderíamos utilizar os caracteres semi-gráficos, mostrados a seguir:

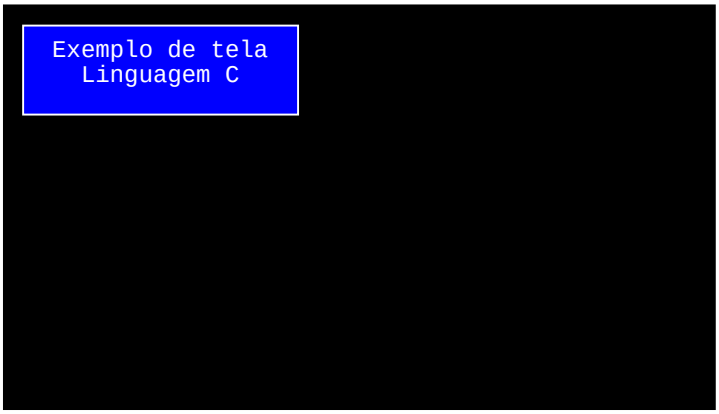


Os caracteres apresentados acima não existem no teclado, para extraí-los devemos manter pressionada a tecla ALT e digitar o código correspondente utilizando o teclado reduzido (teclado numérico). Por exemplo, para extrair o caracter do canto direito superior da moldura basta pressionarmos a tecla ALT e digitarmos 218, soltando a tecla ALT em seguida. Com este processo obteremos um programa com o seguinte aspecto:

```
textcolor(15);
textbackground(1);
gotoxy(1,1); cprint("                    ");
gotoxy(1,2); cprint("  Exemplo de tela   ");
gotoxy(1,3); cprint("    Linguagem C     ");
gotoxy(1,4); cprint("                    ");
```

Note que, durante a digitação, os caracteres aparecerão unidos, dando um efeito de linhas contínuas.

Neste caso, o efeito será o seguinte:



Podemos desta forma obter telas com um aspecto bastante profissional.

Função *scanf() - (scan formated) -* entrada formatada

Objetivo: Permite a leitura formatada de informações via teclado.
Sintaxe.: `scanf(const_char,[lista_var])`

Exemplos:

Algoritmo Inter-S	Linguagem C
Receba vl	`scanf("%d",&vl);`
Receba "Digite o valor",vl	`printf("Digite o valor");scanf("%d",&vl);`

Note que vl é uma variável numérica, sendo assim, dentro da constante alfanumérica const_char , devemos indicar o tipo de dado que a variável receberá, através dos caracteres %d. Estes caracteres são chamados de controle, pois indicam ao compilador qual é o formato do dado que será recebido (lembre-se que as funções de entrada *printf* e *cprintf* também exigem esses caracteres de controle).
No segundo exemplo acima, observe que utilizamos um texto que será escrito na tela antes do recebimento do valor. Em Inter-S a instrução é feita através do comando **Receba**, informando-se entre aspas o texto a ser apresentado na tela e o nome da variável que receberá o conteúdo digitado, separando-os com vírgula (,). Em linguagem C este comando deve ser feito em duas etapas: com a função *printf* (ou *cprintf*) colocamos o texto que deverá aparecer na tela e em seguida utilizamos a função *scanf* para receber o conteúdo da variável.

Resumo

Vimos até agora as funções para entrada e saída de dados utilizadas no modo texto. Com elas podemos construir telas sofisticadas utilizando os recursos de cores, posicionamento do cursor para escrever ou receber informações a partir de qualquer linha ou coluna da tela, etc.
Veja no quadro sinóptico as funções estudadas:

Quadro sinóptico de funções		
Função	**Objetivo**	**Exemplo**
printf()	Saída formatada	printf("Ola pessoal");
cprintf	Saída formatada em cores	cprintf("Ola pessoal");
textcolor()	Definição da cor do texto	textcolor(9);
textbackground()	Definição da cor do segundo plano	textbackground(4);
gotoxy()	Posicionamento do cursor	gotoxy(10,5);
clrscr()	Apaga a tela	clrscr();
scanf()	Entrada formatada	scanf("%d",valor1);

Juntando tudo

Para ilustrar a aplicação das funções estudadas, vamos partir de um exercício resolvido que solicita dois dados pelo teclado e processa a soma dos mesmos, e então, anexaremos a ele algumas "benfeitorias" que o deixarão com um visual mais interessante.

1º passo: Programa original

Algoritmo em Inter-S	Programa fonte em Linguagem C
// Programa Soma.ALG // Autor: Vichinsky // Data: 01/01/2000 **Rotina** **Declare** N1, N2, Soma **Numericos** **Receba** "Digite o primeiro número", N1 **Receba** "Digite o segundo Número", N2 Soma = N1 + N2 **Escreva** "Soma", Soma **FimRotina**	/* Programa SOMA.C Autor: Vichinsky Data.: 01/01/2000 */ #include <stdio.h> #include <conio.h> main() { int n1, n2, soma; printf("Digite o 1º número");scanf("%d",&n1); printf("Digite o 2º número");scanf("%d",&n2); soma = n1 + n2; printf("Soma %d",soma); }

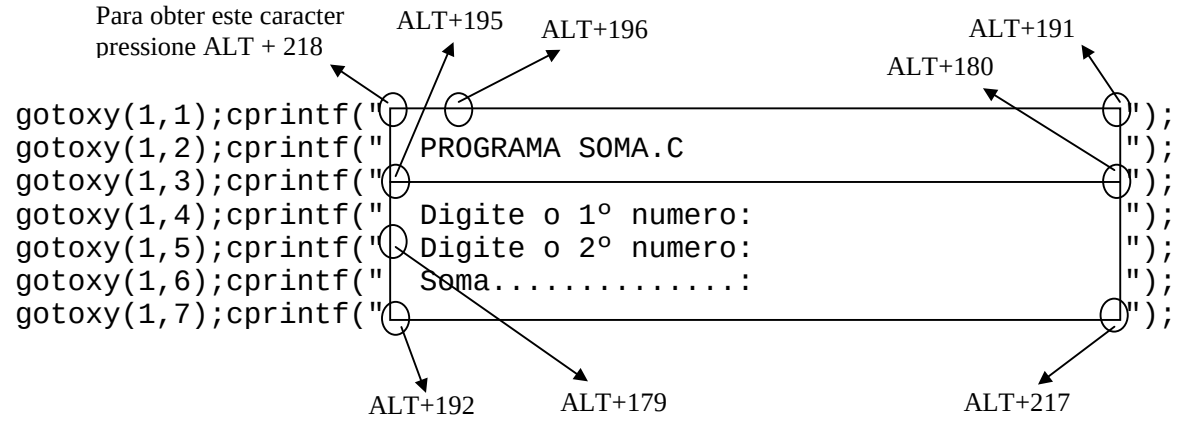
Encontra-se aqui o algoritmo e o código fonte do programa original SOMA.C. Se você ainda não o digitou, faça-o agora.
Nos próximos passos não mais apresentaremos o algoritmo em Inter-S, pois a lógica do programa será mantida. Incluiremos apenas as funções para melhorar o visual da aplicação, e isto não altera a lógica, considera-se apenas um trabalho de embelezamento ou "perfumaria".

2º passo: Colocação de uma moldura
A primeira coisa que iremos fazer é uma moldura que envolva as informações de tela. Para isto teremos que usar a função de limpeza (clrscr) e a função de posicionamento do cursor (gotoxy). Usaremos também a função cprintf em substituição à printf.

```
/* Programa SOMA.C
   Autor: Vichinsky
   Data.: 01/01/2000
*/
#include <stdio.h>
#include <conio.h>

main() {
  int n1, n2, soma;
  clrscr();
  gotoxy(1,1);cprintf("┌──────────────────────────────────────────────────────────┐");
  gotoxy(1,2);cprintf("│ PROGRAMA SOMA.C                                          │");
  gotoxy(1,3);cprintf("├──────────────────────────────────────────────────────────┤");
  gotoxy(1,4);cprintf("│ Digite o 1º numero:                                      │");
  gotoxy(1,5);cprintf("│ Digite o 2º numero:                                      │");
  gotoxy(1,6);cprintf("│ Soma.............:                                       │");
  gotoxy(1,7);cprintf("└──────────────────────────────────────────────────────────┘");
  gotoxy(23,4);scanf("%d",&n1);
  gotoxy(23,5);scanf("%d",&n2);
  soma = n1 + n2;
  gotoxy(23,6);cprintf("%d",soma);
}
```

Note que inicialmente é montada uma moldura envolvendo as informações de tela: identificação do programa e nome dos campos a serem preenchidos. Utilizamos a função *clrscr* para a limpeza de tela e em seguida um conjunto de instruções contendo 7 funções *gotoxy* que têm o objetivo de posicionar o cursor na linha e coluna desejada para que a informação seja exibida através da função *cprintf*. Para a montagem da moldura deve-se empregar os caracteres semi-gráficos (aqueles que não existem no teclado) mantendo-se pressionada a tecla ALT e digitando-se o código correspondente :



Observe também que antes do recebimento dos dados com a função scanf, devemos posicionar o cursor na coluna e linha desejadas:

```
  gotoxy(23,4);scanf("%d",&n1);
  gotoxy(23,5);scanf("%d",&n2);
```

O conteúdo de n1 será solicitado na linha 4 a partir da coluna 23. O conteúdo de n2 será solicitado na linha 5 a partir da coluna 23.

3º passo: Colocação de cores

A colocação de cores é muito simples: basta definirmos as cores do texto e segundo plano através das funções *textcolor* e *textbackground*. Veja como ficará o programa:

```
/* Programa SOMA.C
   Autor: Vichinsky
   Data.: 01/01/2000
*/
#include <stdio.h>
#include <conio.h>

main() {
  int n1, n2, soma;
  clrscr();
  textcolor(15);textbackground(1);
  gotoxy(1,1);cprintf("
  gotoxy(1,2);cprintf("
  gotoxy(1,3);cprintf("
  gotoxy(1,4);cprintf("
  gotoxy(1,5);cprintf("
  gotoxy(1,6);cprintf("
  gotoxy(1,7);cprintf("
  gotoxy(23,4);scanf("%d",&n1);
  gotoxy(23,5);scanf("%d",&n2);
  soma = n1 + n2;
  gotoxy(23,6);cprintf("%d",soma);
}
```

```
 PROGRAMA SOMA.C
 Digite o 1º numero:
 Digite o 2º numero:
 Soma.............:
```

Definimos o texto branco (cor nº 15) e o fundo azul (cor nº 1). As funções de cores foram colocadas após a função de limpeza de tela (*clrscr*), mas você poderá colocá-las antes. Desta forma toda a tela ficará azul, e não apenas a área da moldura. Faça um teste!

Note que após a definição todas a informações escritas com a função *cprintf* assumirão as cores especificadas. Mesmo após o programa ser encerrado estas cores serão mantidas.

CAPÍTULO III
Trabalhando com tela em modo gráfico

Comandos para utilização da tela em modo gráfico / funções da biblioteca gráfica.
Neste capítulo estudaremos os comandos e funções necessárias para a elaboração de programas que trabalhem com telas no modo gráfico.

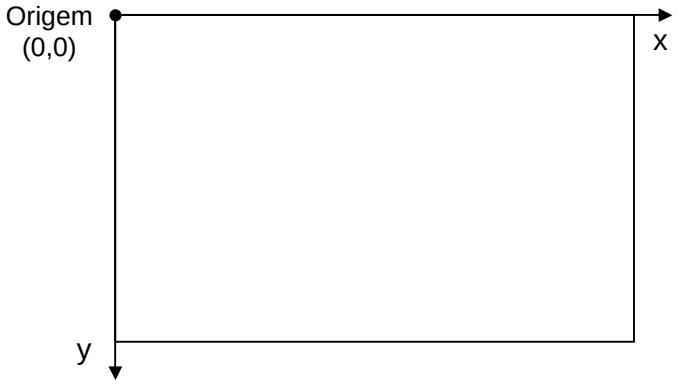
No capítulo anterior estudamos as funções de tela em modo texto. Vimos que a tela é composta por uma matriz de 80 colunas por 25 linhas (em seu estado padrão), e que podemos exibir apenas os caracteres da tabela ASCII. Quando você trabalhar com o MS-DOS (prompt do MS-DOS) observe que os caracteres (letras, números, sinais de pontuação, etc) são formados por um conjunto de pontos. Por exemplo, a letra A, se ampliada, terá o seguinte aspecto:

Cada um dos pontos que compõe um caracter é chamado de *pixel*, e no modo texto nós não temos controle sobre eles, pois são colocados automaticamente quando pedimos a exibição do caracter. É por este motivo que não conseguimos obter desenhos ou figuras diferentes no modo texto.

Como podemos, então, manipular os pixels dentro de um programa para criar desenhos?
A resposta é muito simples: trabalhando no modo gráfico!

Existem várias formas para se trabalhar com telas gráficas na linguagem C, algumas delas requerem conhecimentos técnicos mais aprofundados para que o programador construa sua própria biblioteca de funções gráficas, porém, para as nossas atividades, vamos trabalhar com os recursos oferecidos pelas bibliotecas padrões da linguagem (aquelas que já vêm prontas).

A tela no modo gráfico não trabalha com linhas e colunas como acontece em modo texto, mas sim com pontos horizontais e verticais, constituindo uma matriz. Os pontos, ou pixels (termo técnico utilizado), são referenciados através de dois eixos: x e y, assim como no plano cartesiano:



Representação da tela gráfica. Observe que o ponto origem (coordenada 0,0) encontra-se no canto esquerdo superior do vídeo. O eixo y apresenta uma escala crescente de cima para baixo (não existem negativos)

A quantidade de pontos suportados por uma tela gráfica depende do tipo de adaptador instalado no computador. Os adaptadores (placas de vídeo) mais comuns são: CGA (Adaptador Gráfico Colorido), EGA (Adaptador Gráfico Estendido) e VGA (Matriz Gráfica de Vídeo). Os computadores atuais são equipados, em sua grande maioria, com adaptadores VGA.
Não iremos nos aprofundar nestes detalhes técnicos, mas é importante que você saiba qual é o adaptador gráfico instalado no computador que executará os seus programas, para que reconheça a quantidade de pontos (resolução) suportada por ele.

Para que seus programas acessem a tela gráfica, você deve informar o tipo de adaptador utilizado e o modo de resolução, pois cada adaptador pode trabalhar basicamente com 3 modos de resolução (baixa, média ou alta). Quanto maior a resolução maior o número de pontos suportados.
Você deve estar ansioso para saber como colocar estas informações no programa. Muito bem, o primeiro passo é incluir uma referência ao arquivo *graphics.h* no cabeçalho de seu programa:

```
#include <graphics.h>
```

Em seguida, dentro da rotina principal (main), você deve inserir uma seqüência de instruções para fazer a iniciação do modo gráfico. Estas instruções devem aparecer logo após às declarações das variáveis e constantes utilizadas:

```
int gdriver = VGA, gmode = VGAHI, errorcode;
initgraph(&gdriver, &gmode, "");
errorcode = graphresult();
if (errorcode != grOk) {
  printf("Erro: %s\n", grapherrormsg(errorcode));
  exit(1);
}
```

Parece coisa de louco, não acha? Mas não se assuste, as instruções acima são de uma simplicidade muito grande, pois estamos utilizando algumas funções previamente preparadas pelo fabricante do Turbo C. Vamos aos esclarecimentos:

- Inicialmente estamos declarando duas constantes (*gdriver* e *gmode*) e uma variável *errorcode* :

  ```
  int gdriver = VGA, gmode = VGAHI, errorcode;
  ```

  A constante *gdriver* está recebendo o conteúdo de *VGA*, que na realidade também é uma constante definida dentro do arquivo *graphics.h*, que tem como conteúdo o código interno que representa o adaptador gráfico VGA.
  A constante *gmode* está recebendo o conteúdo de *VGAHI* (constante que contém o código que representa o modo de resolução alta - VGA High)
  A variável *errorcode* está reservada para receber um código de situação, que será analisado nas instruções subseqüentes.

- Após as declarações das constantes e variáveis, temos a seguinte instrução:

  ```
  initgraph(&gdriver, &gmode, "");
  ```

  Estamos aqui chamando a função *initgraph*, que tem o objetivo de colocar a tela em modo gráfico de acordo com as definições entre parênteses. Neste caso estamos informando que o adaptador gráfico é o VGA (definido em *gdriver*) e que a resolução é alta, ou seja, resolução VGA High (definida em *gmode*). Observe também que foram colocadas duas aspas seguidas ("") como último parâmetro da função. Este parâmetro determina o caminho para que o compilador encontre os drivers do adaptador, mas não há necessidade de informá-lo, basta apenas abrir e fechar aspas.

- Após o processo de iniciação do modo gráfico, devemos verificar se a ação foi bem sucedida, ou seja, se a função *initgraph* conseguiu reconhecer o adaptador e o modo de resolução definidos. Para esta verificação devemos atribuir à variável *errorcode* o resultado gerado pela função *graphresult* :

  ```
  errorcode = graphresult();
  ```

  A função *graphresult* retorna o resultado da iniciação do modo gráfico através de um código de situação, o qual será avaliado pelas seguintes instruções:

```
if (errorcode != grOk) {
  printf("Erro: %s\n", grapherrormsg(errorcode));
  exit(1);
}
```

Através de uma estrutura de decisão estamos perguntando se o conteúdo da variável *errorcode* é diferente do conteúdo da constante *grOk*: **if (errorcode != grOk)**
A constante *grOk* está definida no arquivo *graphics.h* e contém o código que indica o sucesso da iniciação do modo gráfico. Para perguntar se *errorcode* é diferente utilizamos o operador relacional != (exclamação e sinal de igualdade), que em linguagem C corresponde a "diferente" ou "não igual".
Se o conteúdo de *errorcode* for diferente de *grOk* então houve um erro na iniciação, neste caso o programa deve ser encerrado. Observe que antes do encerramento será colocada uma informação através da função *printf* :

printf("Erro: %s\n", grapherrormsg(errorcode));

Esta instrução escreverá na tela a expressão "Erro" e a descrição do erro encontrado.
Para finalizar o programa estamos utilizando a função **exit(1)**. O número 1 entre parênteses indica que o programa foi encerrado mediante uma ocorrência de erro.

Você percebeu que apesar da aparência assustadora as instruções para se iniciar a tela gráfica são bastante simples: basta saber qual é o adaptador gráfico que estará utilizando e qual o modo de resolução desejado; coloca-se estas informações nas constantes *gdriver* (adaptador) e *gmode* (modo). A tabela a seguir mostra alguns dos adaptadores que você poderá usar e os modos de resolução suportados por eles:

Adaptador	Resolução	Valor de gdriver	Valor de gmode	Quantidade de pontos
EGA	Baixa (Low)	EGA	EGALO	640 x 200
EGA	Alta (High)	EGA	EGAHI	640 x 350
VGA	Baixa (Low)	VGA	VGALO	640 x 200
VGA	Media (Med)	VGA	VGAMED	640 x 350
VGA	Alta (High)	VGA	VGAHI	640 x 480

Muito bem! Com o conteúdo teórico visto até agora temos como iniciar a construção de um programa gráfico, mas do que adianta acessar a tela gráfica se ainda não sabemos como enviar informações para ela?
Para começar vamos construir um programa que desenhe um círculo amarelo no centro da tela, considerando o adaptador VGA no modo de alta resolução (VGAHI):

```
/* Programa  GRAF1.C - Desenha um círculo amarelo */
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
main(){
   int gdriver = VGA, gmode = VGAHI, errorcode;
   initgraph(&gdriver, &gmode, "");
   errorcode = graphresult();
   if (errorcode != grOk) {
      printf("Erro: %s\n", grapherrormsg(errorcode));
      exit(1);
   }
   setcolor(14); circle(320,240,100);
   getch();
   closegraph();
   exit(0);
 }
```

Explicações

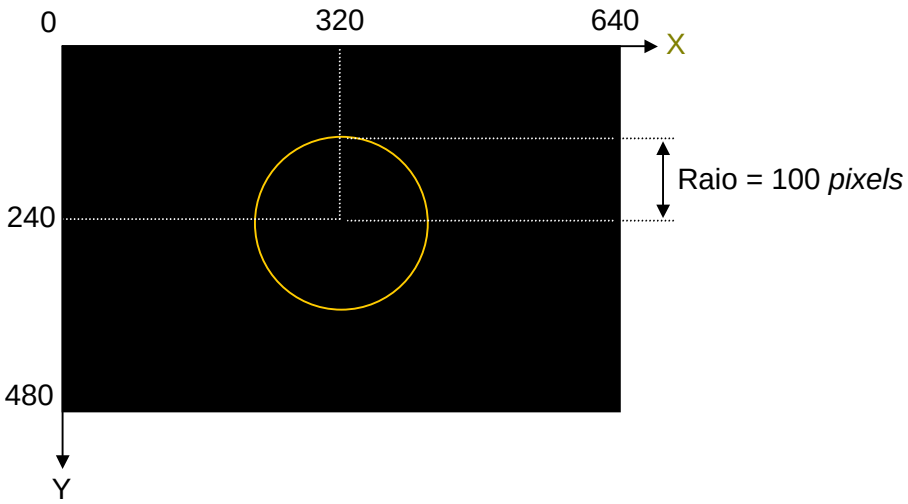
Na tabela abaixo estamos reproduzindo o programa GRAF1.C com as devidas explicações para cada uma das linhas de instruções:

Linhas de instruções	Explicações
/* Programa GRAF1.C -Desenha um círculo */	Linha de identificação (comentário)
#include <graphics.h>	Referência ao arquivo graphics.h (biblioteca)
#include <stdio.h>	Referência ao arquivo stdio.h (biblioteca)
#include <conio.h>	Referência ao arquivo conio.h (biblioteca)
main( ) {	Início da rotina principal
int gdriver = VGA, gmode = VGAHI, errorcode;	Declaração das constantes e variáveis
initgraph(&gdriver, &gmode, "");	Iniciação do modo gráfico
errorcode = graphresult();	Obtenção do resultado da iniciação
if (errorcode != grOk) {	Verifica se houve erro na iniciação
printf("Erro: %s\n", grapherrormsg(errorcode));	Se houve erro escreve uma mensagem na tela e...
exit(1);	...encerra o programa
}	Finaliza a verificação
setcolor(14); circle(320,240,100);	Ajusta a cor para amarelo (14) e desenha o círculo
getch();	Gera uma pausa até que uma tecla seja pressionada
closegraph();	Fecha o modo gráfico
exit(0);	Encerra o programa normalmente
}	Fim da rotina principal

Este programa, simplesmente, desenha um círculo no centro da tela gráfica. A instrução responsável pelo desenho é:

```
setcolor(14); circle(320,240,100);
```

Através da função *setcolor* determinamos a cor do desenho, que neste caso é amarelo (cor número 14). Em seguida, através da função *circle*, desenhamos o círculo. Observe que entre os parênteses da função existem três parâmetros: 320, 240, 100. Os dois parâmetros iniciais indicam a coordenada da tela que será o centro do círculo (x=320 e y=240) e o terceiro parâmetro indica o tamanho do raio medido em pontos gráficos (*pixels*). Observe a representação abaixo:



Note que o centro do círculo encontra-se justamente no centro da tela, ou seja, na posição 320 do eixo X e 240 do eixo Y (lembre-se que no padrão VGA de alta resolução temos 640 pontos no eixo X e 480 pontos no eixo Y, a coordenada central do círculo utiliza justamente a metade desses valores). Para aumentar ou diminuir o tamanho do círculo basta alterar o valor do raio, aumentando-o ou diminuindo-o respectivamente.

Com este pequeno programa exemplo já temos uma noção de como utilizar o modo gráfico em nossas aplicações. Porém, antes de prosseguirmos, vejamos um resumo das funções gráficas que iremos utilizar em nossos futuros programas.

Resumo das principais funções gráficas

cleardevice()	Limpa a tela gráfica.
setcolor(c)	Define a cor das imagens (c = código da cor, de 0 a 15).
setfillstyle(p,c)	Define a cor de preenchimento dos retângulos e elipses (p=padrão de preenchimento; c=código da cor).
bar(x1,y1,x2,y2)	Desenha um retângulo preenchido com a cor definida pela função setfillstyle, a partir do ponto x1-y1 até a ponto x2-y2.
fillellipse(x,y,r1,r2)	Desenha uma elipse preenchida com a cor definida pela função setfillstyle, com centro na coordenada x-y, com tamanho do maior raio definido por r1 e menor raio definido por r2.
circle(x,y,r)	Desenha um círculo sem preenchimento, com centro na coordenada x-y e raio igual a r.
rectangle(x1,y1,x2,y2)	Desenha um retângulo sem preenchimento, a partir do ponto x1-y2 até o ponto x2-y2.
line(x1,y1,x2,y2)	Traça uma linha iniciando no ponto x1-y1 até o ponto x2-y2.
putpixel(x,y,c)	Plota um ponto na coordenada x-y (c=código da cor, de 0 a 15).
sprintf(s,f,v)	Envia para a variável string (s) um texto formatado (f) que pode conter valores de variáveis (v).
outtextxy(x,y,s)	Escreve na tela gráfica, a partir do ponto x-y, o conteúdo definido em s.
closegraph()	Fecha o modo gráfico e volta ao modo texto.

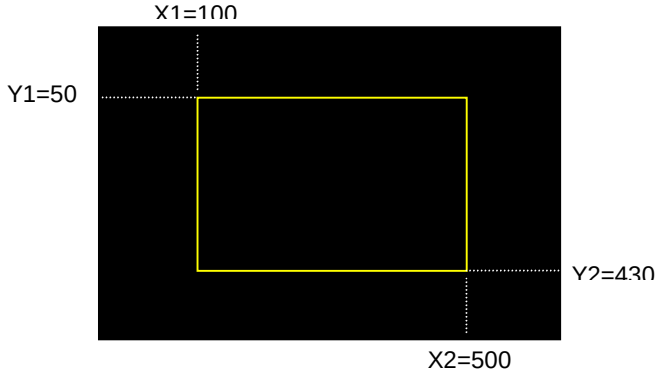
Nota: As referências em azul indicam os valores dos parâmetros requeridos pela função. Estes valores podem ser constantes ou variáveis. Por exemplo, para desenhar um retângulo a partir da coordenada 100,50 até a coordenada 500,430 podemos usar uma das seguintes instruções:

```
rectangle(100,50,500,430);
```

ou

```
x1=100; y1=50; x2=500; y2=430;
rectangle(x1,y1,x2,y2);
```

Em ambos os casos teremos o seguinte resultado:



Textos em tela gráfica

Você deve estar perguntando: *é possível escrever textos em uma tela gráfica?*
Sim! Mas não com os mesmos métodos utilizados em telas no modo texto. As funções *printf* e *cprintf* (nossas velhas conhecidas) não funcionam adequadamente em modo gráfico. Devemos substitui-las pela função *sprintf* em conjunto com a função *outtextxy*. Estas funções estão relacionadas no resumo apresentado na página anterior.
A função *sprintf* não escreve nada na tela, simplesmente prepara uma variável do tipo caracter que posteriormente será escrita através da função *outtextxy*. Podemos também utilizar a função *outtextxy* isoladamente.Para ilustrar este processo imagine que você queira escrever o texto *"Ola amigos da Linguagem C"* a partir da coordenada 10,20 (x=10 e y=20), na cor azul, e em seguida exibir o conteúdo de uma variável. Observe o programa que realiza esta tarefa:

```
/* Programa  GRAF2.C - Escrita em tela gráfica */
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
main(){
   int var;
   char mensagem[80];
   int gdriver = VGA, gmode = VGAHI, errorcode;
   initgraph(&gdriver, &gmode, "");
   errorcode = graphresult();
   if (errorcode != grOk) {
      printf("Erro: %s\n", grapherrormsg(errorcode));
      exit(1);
   }
   setcolor(1);
   outtextxy(10,20,"Ola amigos da Linguagem C");
   var=35;
   sprintf(mensagem, "Valor da variável var: %d", var);
   outtextxy(10,50,mensagem);
   getch();
   closegraph();
   exit(0);
 }
```

Observe que para escrever o texto *"Ola amigos da Linguagem C"* utilizamos a função *outtextxy*:

```
outtextxy(10,20,"Ola amigos da Linguagem C");
```

Neste caso não houve necessidade de se utilizar a função *sprintf*, pois a informação a ser escrita não contém variáveis. Por outro lado, para exibir o conteúdo da variável *var* é preciso que se prepare a informação através da função *sprintf* :

```
var=35;
sprintf(mensagem, "Valor da variável var: %d", var);
outtextxy(10,50,mensagem);
```

Note que a função *sprintf* coloca na variável caracter *mensagem* o texto *"Valor da variável"* , agregando o conteúdo da variável *var*. Em seguida utilizamos a função *outtextxy* para exibir o conteúdo de *mensagem* a partir da coordenada 10,50. (Observe que a variável *mensagem* é declarada no início da rotina através da instrução **`char mensagem[80];`**).

Alguns programas exemplos

Para que você assimile melhor os comandos gráficos apresentados, vamos expor alguns programas exemplos.

Programa exemplo 1: Desenho livre: desenha linhas na tela com o uso das setas

O programa abaixo, após iniciar o modo gráfico e preparar as variáveis de trabalho, entra em uma estrutura de repetição (bloco entre as instruções *volta:* e *goto volta;*) e fica aguardando o pressionamento de uma tecla (instrução *op=getch();*). De acordo com a tecla pressionada é acionado um determinado bloco de instruções. A seleção do bloco é feita por comandos *if* que verificam o código da tecla pressionada. As teclas programadas são:

Tecla	Código	Ação
Seta p/ cima	H	Diminui 1 da variável y e coloca um ponto na nova coordenada.
Seta p/ baixo	P	Soma 1 na variável y e coloca um ponto na nova coordenada.
Seta p/ direita	M	Soma 1 da variável x e coloca um ponto na nova coordenada.
Seta p/ esquerda	K	Diminui 1 da variável x e coloca um ponto na nova coordenada.
Dígito 0 (zero)	0	Ajusta a cor para 0 (preto)
Dígito 1 (um)	1	Ajusta a cor para 1 (azul)
Dígito 2 (dois)	2	Ajusta a cor para 2 (verde)
Dígito 3 (três)	3	Ajusta a cor para 3 (ciano)
Asterisco	*	Encerra o programa

```
/* Programa LIVRE.C - Desenho livre com uso das setas */
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
main(){
   int i,x,y,cor;
   char op;
   char mensagem[80];
   int gdriver = VGA, gmode = VGAHI, errorcode;
   initgraph(&gdriver, &gmode, "");
   errorcode = graphresult();
   if (errorcode != grOk) {
      printf("Erro: %s\n", grapherrormsg(errorcode));
      exit(1);
   }
   cleardevice();
   x=320; y=240; cor=1;
   volta:
      sprintf(mensagem, "X= %d Y= %d Cor= %d", x,y,cor);
      outtextxy(10,400,mensagem);
      op=getch();
      setcolor(0); outtextxy(10,400,mensagem);
      setcolor(15);
      if (op=='H'){
         if (y>0) y=y-1;
         putpixel(x,y,cor);
      }
      if (op=='P'){
         if (y<400) y=y+1;
         putpixel(x,y,cor);
      }
      if (op=='M'){
         if (x<639) x=x+1;
         putpixel(x,y,cor);
      }
```

```
      if (op=='K'){
         if (x>0) x=x-1;
         putpixel(x,y,cor);
      }
      if (op=='0'){cor=0;}
      if (op=='1'){cor=1;}
      if (op=='2'){cor=2;}
      if (op=='3'){cor=3;}
      if (op=='*'){closegraph(); exit(0);}
   goto volta;
}
```

Programa exemplo 2: Cores e figuras: Demonstra o uso de cores e desenho de figuras

O programa abaixo, após iniciar o modo gráfico e preparar as variáveis de trabalho, faz uma demonstração de cores, exibindo todas as cores possíveis neste modo. Em seguida apresenta um menu que permite a seleção de 3 opções. Analise-o:

```
/* Programa: CORES.C - Demonstração das cores */
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
main(){
   int i;
   char op;
   char mensagem[80];
   int gdriver = VGA, gmode = VGAHI, errorcode;
   initgraph(&gdriver, &gmode, "");
   errorcode = graphresult();
   if (errorcode != grOk) {
      printf("Erro: %s\n", grapherrormsg(errorcode));
      exit(1);
   }
   cleardevice();
   setcolor(1); outtextxy(10, 1, "Cor 1");   circle(70,5,4);
   setcolor(2); outtextxy(10, 16, "Cor 2");  circle(70,20,4);
   setcolor(3); outtextxy(10, 31, "Cor 3");  circle(70,35,4);
   setcolor(4); outtextxy(10, 46, "Cor 4");  circle(70,50,4);
   setcolor(5); outtextxy(10, 61, "Cor 5");  circle(70,65,4);
   setcolor(6); outtextxy(10, 76, "Cor 6");  circle(70,80,4);
   setcolor(7); outtextxy(10, 91, "Cor 7");  circle(70,95,4);
   setcolor(8); outtextxy(10, 106, "Cor 8"); circle(70,110,4);
   setcolor(9); outtextxy(10, 121, "Cor 9"); circle(70,125,4);
   setcolor(10);outtextxy(10, 136, "Cor 10");circle(70,140,4);
   setcolor(11);outtextxy(10, 151, "Cor 11");circle(70,155,4);
   setcolor(12);outtextxy(10, 166, "Cor 12");circle(70,170,4);
   setcolor(13);outtextxy(10, 181, "Cor 13");circle(70,185,4);
   setcolor(14);outtextxy(10, 196, "Cor 14");circle(70,200,4);
   setcolor(15);outtextxy(10, 211, "Cor 15");circle(70,215,4);
   outtextxy(10,250,"Tecle algo para continuar");
   getch();
   menu:
      cleardevice();
      setcolor(14);
      outtextxy(250, 5, "Menu de opcoes");
      setcolor(3);
      outtextxy(220, 20, "[1] Desenho de circulos");
```

```
      outtextxy(220, 30, "[2] Desenho de Linhas");
      outtextxy(220, 40, "[3] Fim");
      outtextxy(220, 60, "Digite sua opcao");
      op=getch();
      cleardevice();
      sprintf(mensagem, "Opcao selecionada = %c", op);
      outtextxy(10,10,mensagem);
      if(op=='1'){
         setcolor(1); circle(320,240,150);
         setcolor(2); circle(100,200,80);
         setcolor(14);circle(400,240,120);
         setcolor(6); circle(300,300,100);
         getch();
         cleardevice();
      }
      if (op=='2'){
         setcolor(14);line(1,1,600,300)  ; getch();
         setcolor(1) ;line(100,1,560,300); getch();
         setcolor(2) ;line(1,300,600,5)  ; getch();
         setcolor(3) ;line(600,1,1,100)  ; getch();
         setcolor(4) ;line(300,300,1,1)  ; getch();
         setcolor(5) ;line(550,1,1,300)  ; getch();
         setcolor(6) ;line(600,1,1,400)  ; getch();
      }
      if (op=='3'){
         closegraph(); exit(0);
      }
   goto menu;
}
```

Programa exemplo 3: Figuras: Demonstra o desenho de figuras

Este exemplo é o mais simples de todos. Mostra algumas figuras desenhadas através das funções gráficas estudadas:

```
/* Programa FIGURAS.C - Desenho de figuras */
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
main(){
   int gdriver = VGA, gmode = VGAHI, errorcode;
   initgraph(&gdriver, &gmode, "");
   errorcode = graphresult();
   if (errorcode != grOk) {
      printf("Erro: %s\n", grapherrormsg(errorcode));
      exit(1);
   }
   setfillstyle(1,14);          /* Padrão e cor de preenchimento */
   bar(10,10,50,50);            /* Desenha uma barra preenchida  */
   fillellipse(200,200,50,30);  /* Desenha uma elipse preenchida */
   circle(300,300,50);          /* Desenha um círculo            */
   rectangle(400,100,500,110);  /* Desenha um retângulo          */
   line(100,1,300,400);         /* Desenha uma linha             */
   putpixel(320,300,12);        /* Desenha um ponto (pixel)      */
   getch();                     /* Aguarda uma tecla qualquer    */
   closegraph();                /* Fecha o modo gráfico          */
   exit(0);                     /* Encerra o programa            */
}
```

Atividades complementares

Digite os programas apresentados neste capítulo (GRAF1.C, GRAF2.C, LIVRE.C, CORES.C e FIGURAS.C). Veja o funcionamento dos mesmos e faça alterações a seu gosto, como por exemplo: mudar as cores, alterar o tamanho das figuras, modificar os textos, etc. É importante que você compreenda o funcionamento das funções estudadas, e a prática é o melhor caminho para isso.

***Observação**: Para que um programa gráfico possa ser executado, é preciso que o arquivo EGAVGA.BGI esteja presente na mesma pasta (diretório) onde se encontra o programa. Este arquivo contém um conjunto de funções específicas para que o programa reconheça o modo gráfico (este arquivo faz parte do pacote Turbo C). No nosso caso não teremos problema, pois o arquivo EGAVGA.BGI está presente na pasta TC201, mas se você quiser copiar um programa gráfico para um computador que não possua o Turbo C, não esqueça de copiar também o arquivo EGAVGA.BGI.*

CAPÍTULO IV
Arquivos de dados

Funções de biblioteca para a manipulação de arquivos em disco.
Neste capítulo estudaremos as funções básicas da linguagem C para a manipulação de dados em arquivos gravados em disco.

Os programas vistos até agora trabalham com informações que são armazenadas em variáveis de memória. As variáveis conservam seus conteúdos somente enquanto o programa está sendo executado, quando o programa é finalizado, todos os dados manipulados por ele são perdidos.
Porém, existem situações em que necessitamos armazenar as informações de forma permanente, para que não sejam perdidas quando o programa for encerrado. Vamos imaginar que você queira construir um programa para armazenar os dados de seus amigos - uma agenda telefônica de amigos. Para este feito será necessário o armazenamento permanente das informações, pois você não deseja perdê-las quando o programa for encerrado. Como fazer isto?
Bem, podemos começar com uma rotina que recebe as informações dos amigos:

```
/* Programa AGENDA.C */
#include<stdio.h>
#include<conio.h>
main(){
   int codigo;
   char nome[30];
   char telefone[15];
   char resp;
   inicio:
     clrscr();
     printf("Digite o codigo do amigo..: ");scanf("%d",&codigo);
     printf("Digite o nome do amigo....: ");scanf("%s",nome);
     printf("Digite o telefone do amigo: ");scanf("%s",telefone);
     printf("Continua (S/N)? ");scanf("%s",&resp);
     if (resp=='N' ¦¦ resp=='n') {
        exit(0);
     }
   goto inicio;
}
```

O programa acima (AGENDA.C) começa com a declaração de 4 variáveis: *codigo*, *nome*, *telefone* e *resp*. Em seguida existe um rótulo denominado *inicio*, que indica o ponto onde se inicia o processo de recebimento dos dados. Os dados *codigo*, *nome* e *telefone* do amigo são recebidos pelo teclado através das funções **scanf**. Em seguida a mensagem *"Continua (S/N)?"* é mostrada na tela através da função **printf** e o dado *resp* é recebido pelo teclado. Se o conteúdo de *resp* for N ou n o programa será encerrado, caso contrário haverá um desvio para o rótulo *inicio* (**goto inicio;**), repetindo-se o processo de recebimento dos dados.
Se você digitar e compilar esta rotina perceberá que o funcionamento dela é perfeito: ela receberá as informações pelo teclado e as armazenará nas variáveis declaradas. Mas existirá um problema: as variáveis conservarão seus conteúdos até que outro conteúdo seja atribuído a elas, ou seja, esta rotina **não** faz o armazenamento permanente dos dados recebidos, portanto, não serve para absolutamente nada!

Qual a saída para que os dados digitados sejam armazenados permanentemente?

Armazenando-os em um arquivo de dados!

Um arquivo de dados é um conjunto de informações armazenado em um meio magnético ou óptico (disquete, disco rígido, fita, Cd, etc.) onde permanecerá gravado e disponível para quaisquer futuras utilizações.

Para ilustrar este conceito, vamos imaginar que a nossa agenda de amigos tenha as seguintes informações:

Agenda de amigos		
Código	**Nome**	**Telefone**
1	Antonio Laurentino Miguel	1234-5678
2	Bianco Branco de Assis Moreira	8765-4321
3	Juarez João Columbiano Mattos	9999-7777
4	Pedro Pedreira Rocha	0001-1110
5	Zé Mané	1000-1234

Estas informações, da forma que estão estruturadas, caracterizam um arquivo de dados. Podemos através disto tirar algumas definições:

- Um arquivo de dados deve possuir um nome. No nosso caso o nome é "Agenda de amigos", mas em linguagem C o nome deve possuir no máximo 8 (oito) caracteres e um sufixo com no máximo 3 (três), vamos usar "AGENDA.DAT" (o sufixo deve aparecer após um ponto final e deve informar o tipo de arquivo; estamos utilizando a expressão DAT, que é uma abreviação de *data* - *dado* em inglês - mas poderíamos utilizar qualquer sufixo desejado).
- Um arquivo de dados deve possuir uma estrutura bem definida, observe as linhas e colunas, que nos dão a idéia de uma matriz ou tabela. Cada uma da linhas armazena os dados de um amigo, e cada coluna indica um dado (código, nome ou telefone).
- Tecnicamente chamamos as linhas de *registros* e as colunas de *campos*.
- Todos os registros possuem os mesmos campos e, portanto, o mesmo tamanho.

Com estas definições básicas, podemos concluir que:

- Um arquivo de dados é um conjunto de informações formado por registros e campos.
- Um registro é um conjunto de campos que armazena os dados de um item do arquivo.
- Um campo é um dado individual do registro.

Já temos uma noção do que é um arquivo de dados, vamos ver agora como fazer com que os dados dos nossos amigos sejam gravados em um arquivo.

Instruções necessárias para a gravação de dados em um arquivo

Primeiro passo: Preparação

Para utilizar um arquivo de dados em um programa, o primeiro passo é definir um nome para o ponteiro do arquivo. Este ponteiro servirá para direcionar as futuras operações de abertura, gravação e leitura. Utilizaremos o nome *arquivo* para o ponteiro (sugestivo, não?), incluindo a seguinte linha de comando no início do programa:

```
FILE *arquivo;
```

Em seguida devemos declarar a estrutura do arquivo, ou seja, devemos definir os campos que o arquivo possuirá. No nosso exemplo do arquivo AGENDA.DAT (agenda de amigos) temos os campos: codigo, nome e telefone, portanto, devemos construir uma estrutura que os defina:

`struct {`	Inicia a definição da estrutura
`  int codigo;`	Declara o campo *codigo* como numérico inteiro
`  char nome[30];`	Declara o campo *nome* como caractere de 30 posições
`  char telefone[15];`	Declara o campo *telefone* como caractere de 15 posições
`} registro;`	Nomeia a estrutura como *registro*

Observe que devemos utilizar o comando *struct* e entre chaves indicar os tipos e os nomes de cada um dos campos do arquivo. Após o fechamento da chave devemos indicar o nome da estrutura, que neste exemplo é *registro*, também bastante sugestivo, pois na realidade a estrutura construída corresponde ao registro do arquivo AGENDA.DAT.

Desta forma o nosso programa AGENDA.C deverá possuir as seguintes linhas iniciais:

```
/* Programa AGENDA.C */
#include<stdio.h>
#include<conio.h>
FILE *arquivo;
struct {
  int codigo;
  char nome[30];
  char telefone[15];
}registro;

main(){
   ...
```

Segundo passo: Abertura

Para ilustrar o processo de abertura de um arquivo de dados, imagine que este arquivo seja um caderno. Muito bem, para que possamos utilizar este caderno (escrever ou ler) devemos primeiramente abri-lo. É justamente isto o que ocorre com um arquivo de dados gravado em um disco: para utilizá-lo devemos antes processar sua abertura!

Porém, às vezes a abertura de um caderno não é tão simples. Imagine, por exemplo, que o caderno não exista, ou que alguém tenha derramado cola sobre ele. Nestes casos a abertura será impossível. Se o caderno não existir, menos mal, compramos um novo e pronto! Mas se o caderno estiver colado, os dados nele escritos estarão perdidos, o jeito será jogá-lo fora e comprar um novo.

Estas situações também podem ocorrer com um arquivo de dados: podemos tentar abrir um arquivo que não exista no disco ou, existindo, esteja danificado. Por este motivo, antes de abrirmos um arquivo de dados devemos verificar se a abertura é possível. Podemos representar a rotina de verificação através do seguinte algoritmo em pseudo-linguagem (observe-o com atenção):

```
Se o arquivo "AGENDA.DAT" não pode ser aberto para leitura e escrita então
   Se o arquivo "AGENDA.DAT" não pode ser criado então
      Escreva "Erro de abertura"
      Interrompa
   FimSe
FimSe
```

Este algoritmo, traduzido em linguagem C, terá o seguinte aspecto:

```
if((arquivo=fopen("AGENDA.DAT","r+b"))==NULL) {
   if((arquivo=fopen("AGENDA.DAT","wb"))==NULL) {
      printf("Erro de abertura");
      exit(1);
   }
}
fclose(arquivo);
```

Vamos tentar entender? Então leia com bastante atenção:

- A primeira linha faz a tentativa de abertura do arquivo "AGENDA.DAT" e ao mesmo tempo pergunta se houve sucesso na operação, comparando o ponteiro *arquivo* com o valor NULL (nulo). Se o ponteiro for nulo então o arquivo não pôde ser aberto por algum problema (ou não existe ou está danificado). A expressão "r+b" solicita a abertura do arquivo para a leitura e escrita.

- A segunda linha só será executada se o arquivo não pôde ser aberto. Observe que esta linha é semelhante à primeira, com a diferença na expressão que define o tipo de abertura: "wb". Na realidade esta instrução faz a tentativa de criação do arquivo e ao mesmo tempo verifica se a criação foi bem sucedida, comparando o ponteiro *arquivo* com o valor NULL (nulo). Se o ponteiro for nulo significa que o arquivo também não pode ser criado, neste caso não há nada mais a fazer, a não ser finalizar o programa.

- A terceira linha, assim como a quarta, só será executada se o arquivo "AGENDA.DAT" não puder ser aberto nem criado. Esta linha exibirá na tela a mensagem "Erro de abertura". Em seguida será executada a quarta linha: *exit(1)*, que processa o encerramento do programa mediante ocorrência de erro.

- A quinta linha apenas encerra a estrutura condicional iniciada na segunda linha.

- A sexta linha, assim como a quinta, apenas encerra a estrutura condicional iniciada na primeira linha.

- A sétima e última linha, que não possui correspondência no algoritmo em pseudo-linguagem, faz o fechamento do arquivo.

Em síntese, a rotina apresentada simplesmente verifica se o arquivo pode ser aberto, criando-o caso ele não exista, ou finalizando o programa caso esteja danificado.

Agora podemos abrir o arquivo definitivamente, com a certeza de que haverá sucesso na operação. Usaremos o seguinte comando:

```
arquivo=fopen("AGENDA.DAT","r+b")
```

A função *fopen* é a responsável pela abertura do arquivo. Devemos informar, entre parênteses, o nome do arquivo e o modo de acesso: "AGENDA.DAT" é o nome e "r+b" determina que o modo de acesso é leitura/escrita, isto é, o arquivo poderá ser lido e escrito (como um caderno).

Veja na página seguinte como ficará o início do programa após a inclusão destas instruções:

```
/* Programa AGENDA.C */
#include<stdio.h>
#include<conio.h>
FILE *arquivo;
struct {
  int codigo;
  char nome[30];
  char telefone[15];
}registro;
main() {
   int wcodigo;
   char wnome[30];
   char wtelefone[15];
   char resp;
   if((arquivo=fopen("AGENDA.DAT","r+b"))==NULL) {
      if((arquivo=fopen("AGENDA.DAT","wb"))==NULL) {
         printf("Erro de abertura");
         exit(1);
      }
   }
   fclose(arquivo);
   arquivo=fopen("AGENDA.DAT","r+b");
   ...
```

Note que nas declarações os nomes das variáveis estão sublinhados. Fizemos este destaque para indicar que os nomes foram alterados para não entrarem em conflito com os nomes dos campos definidos na estrutura do arquivo (*struct*). Agora as variáveis *wcodigo*, *wnome* e *wtelefone* serão as variáveis que receberão os dados pelo teclado, e *codigo*, *nome* e *telefone* serão os campos do arquivo de dados.

Terceiro passo: Escrita dos dados

No primeiro passo realizamos a preparação do ponteiro e a definição da estrutura do arquivo de dados, ou seja, definimos um nome para referenciar o arquivo (nome do ponteiro) e definimos os nomes e tipos dos campos que serão gravados (*codigo*, *nome* e *telefone*).
No segundo passo realizamos a abertura do arquivo de dados.
Agora falta apenas saber como escrever os dados no arquivo. Para isto, devemos receber as informações pelo teclado e transferi-las para o arquivo. Vamos utilizar uma estrutura de repetição que será iniciada com o rótulo início, veja como ficará o algoritmo em Inter-S deste trecho:

```
[inicio]
    Limpa
    Receba "Digite o codigo do amigo..: ", wcodigo
    Receba "Digite o nome do amigo....: ", wnome
    Receba "Digite o telefone do amigo: ", wtelefone
    Codigo = wcodigo
    Nome = wnome
    Telefone = wtelefone
    Grave item[codigo] AGENDA.DAT
    Receba "Continua (S/N)? ", resp
    Se resp = 'N' ou resp = 'n' entao
      Interrompa
    FimSe
Vapara inicio
```

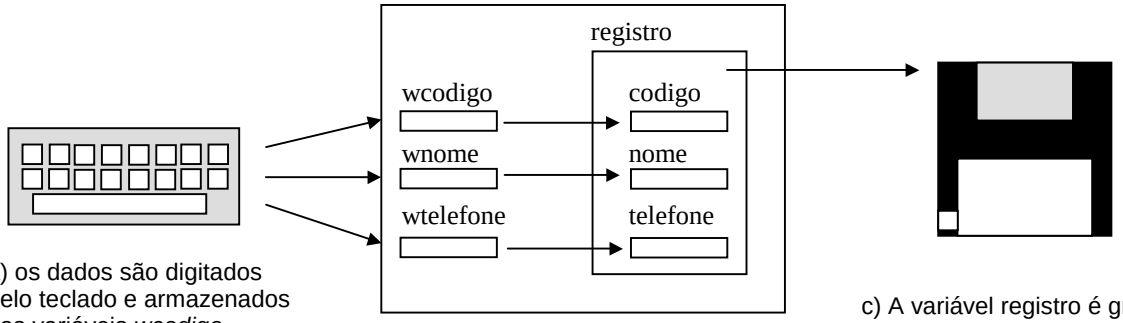
Observe que no algoritmo, após o recebimento dos dados pelo teclado, estamos atribuindo aos campos do registro os conteúdos recebidos pelas variáveis:

```
Codigo = wcodigo
Nome = wnome
Telefone = wtelefone
```

Em linguagem C, para referenciar um campo devemos informar o nome do registro (que no nosso caso é o próprio nome *registro* - definido na estrutura *struct*) seguido por um ponto final e o nome do campo. Exemplo: *registro.codigo*.
Em seguida devemos gravar o registro no arquivo, usamos em Inter-S o seguinte comando:

```
Grave Item[codigo] AGENDA.DAT
```

A figura abaixo ilustra o processo de gravação, observe-a:



a) os dados são digitados pelo teclado e armazenados nas variáveis *wcodigo*, *wnome* e *wtelefone*

b) os dados são repassados para os campos do registro. Na realidade, a variável *registro* é uma estrutura que possui 3 outras variáveis: *codigo*, *nome* e *telefone*.

c) A variável registro é gravada no arquivo em disco, levando consigo as 3 variáveis que a formam.

Implementando este processo de gravação no nosso programa AGENDA.C, teremos o seguinte código final:

```
/* Programa AGENDA.C */
#include<stdio.h>
#include<conio.h>
FILE *arquivo;
struct {
  int codigo;
  char nome[30];
  char telefone[15];
}registro;
main() {
   int wcodigo;
   char wnome[30];
   char wtelefone[15];
   char resp;
   if((arquivo=fopen("AGENDA.DAT","r+b"))==NULL) {
      if((arquivo=fopen("AGENDA.DAT","wb"))==NULL) {
         printf("Erro de abertura");
         exit(1);
      }
   }
   fclose(arquivo);
```

```
   arquivo=fopen("AGENDA.DAT","r+b");
   inicio:
     clrscr();
     printf("Digite o codigo do amigo..: ");scanf("%d",&wcodigo);
     printf("Digite o nome do amigo....: ");scanf("%s",wnome);
     printf("Digite o telefone do amigo: ");scanf("%s",wtelefone);
     registro.codigo = wcodigo;
     strcpy(registro.nome, wnome);
     strcpy(registro.telefone, wtelefone);
     fwrite(&registro, sizeof(registro), 1, arquivo);
     printf("Continua (S/N)? ");scanf("%s",&resp);
     if (resp=='N' ¦¦ resp=='n') {
        fclose(arquivo);
        exit(0);
     }
   goto inicio;
}
```

Observando o programa acima você encontrará algumas funções que, a princípio, nos parecem bastante estranhas (são as funções que destacamos em azul). Vamos analisá-las:

`strcpy(registro.nome, wnome);`

A função *strcpy* faz com que a variável caracter indicada após a abertura do parêntese receba o conteúdo da variável que aparece depois da vírgula. Neste caso *registro.nome* receberá o conteúdo de *wnome*. A instrução equivalente em Inter-S é: **nome = wnome**.
strcpy é exigida apenas para atribuições de variáveis do tipo *string* (caracter), para variáveis numéricas utiliza-se apenas o sinal de igualdade (=), exemplo: **registro.codigo=wcodigo**.

`fwrite(&registro, sizeof(registro), 1, arquivo);`

A função *fwrite* faz a gravação (escrita) dos dados no arquivo. Precisamos informar, entre os parênteses, o nome da variável que contém os dados (precedida do ponteiro &), o tamanho desta variável (sizeof), a quantidade de registros que serão gravados e o nome do ponteiro do arquivo:

`&registro`.....Nome da variável (estrutura) que contém os dados
`sizeof(registro)`.....Tamanho da variável (estrutura) que contém os dados
`1`.....Quantidade de registros a gravar
`arquivo`.....Nome do ponteiro que identifica o arquivo a ser gravado

`fclose(arquivo);`

A função *fclose* realiza o fechamento do arquivo identificado pelo ponteiro entre parênteses.

Concluindo

A rotina de gravação do programa AGENDA.C está completa. Digite o programa e veja o seu funcionamento. Mesmo que tenha digitado corretamente, você notará um grande problema durante a digitação dos dados, a saber:
Quando você digita um espaço no nome do amigo, o programa colocará o texto que vem após o espaço como conteúdo da próxima variável, gerando uma bagunça na lógica do programa. Tente fazer isso: digite Ze Mane como nome do amigo e perceba que o telefone não será solicitado, pois o programa colocará o Ze na variável *wnome* e Mane na variável *wtelefone*.

Como resolver este problema?

Perceba então que o espaço funciona como um "delimitador" da informação digitada. Para sanar este problema, faça com que a função *scanf* reconheça o espaço como um caracter normal.

Para isto devemos alterar a rotina de recebimento, implementando alguns itens. Observe como ela ficará:

```
printf("Digite o codigo do amigo..: ");scanf("%d",&wcodigo);
fflush(stdin);
printf("Digite o nome do amigo....: ");scanf("%30[A-z ]",wnome);
fflush(stdin);
printf("Digite o telefone do amigo: ");scanf("%15s",wtelefone);
fflush(stdin);
```

Devemos colocar a função ***fflush(stdin)*** após as instruções de recebimento (*scanf*). Esta função processa o "descarregamento" da memória do teclado, de maneira que não reste nenhuma informação "perdida", que poderia ser agregada indevidamente ao conteúdo da variável do próximo *scanf*.

Para o recebimento dos dados caracteres estamos utilizando um controle um pouquinho diferente. No programa original estávamos usando a seqüência "%s" para indicar que a variável wnome é do tipo caractere: **scanf("%s",wnome);** , agora estamos usando o controle "%30[A-z]" : **scanf("%30[A-z]",wnome);** . O número 30 informa que o tamanho da variável é de 30 caracteres; as letras A-z entre os colchetes informam que só serão aceitos caracteres de A (maiúsculo) à z (minúsculo), ou seja, durante a digitação do nome do amigo serão aceitas apenas letras maiúsculas e/ou minúsculas; existe ainda um espaço após o z (antes do fechamento do colchete), este espaço indica que também serão aceitos espaços em branco no nome do amigo. Com estas alterações o seu programa poderá receber qualquer nome e sobrenome, inclusive o Zé Mané!

IMPORTANTE: *Espero que neste ponto você já tenha digitado e compilado o programa AGENDA.C, e observado o seu funcionamento com as implementações citadas acima. Mas se você ainda não fez isso, sugiro que o faça antes de prosseguir, pois os conceitos que veremos a partir daqui estarão nele baseados.*

Instruções necessárias para a leitura de um arquivo de dados

Com certeza você notou que o programa AGENDA.C não tem sentido nenhum se não tivermos um programa que faça a leitura do arquivo de dados e que nos mostre os registros gravados. Temos até dúvidas se os registros estão sendo gravados ou não! Bem, esta dúvida até que é fácil sanar, basta você verificar que na pasta onde está o programa, existe agora um arquivo com o nome AGENDA.DAT, este arquivo armazena os dados que você digitou.

Como construir um programa para ler este arquivo?

Primeiramente vamos definir como será esta leitura. Imagine que você queira ver todos os dados gravados, ou seja, que você queira "listar" todos os seus amigos na tela, um a um. Podemos construir um algoritmo em Inter-S para isto:

```
Rotina
  Declare arquivo(AGENDA.DAT) chave(codigo)
  Registro ( codigo Numerico [4]
             nome caractere [30]
             telefone caractere [15])
  Abra AGENDA.DAT
  Topo
  [Inicio]
  Leia Proximo registro Agenda.dat
  Se nao FDA entao
      Escreva "Codigo do amigo", codigo
      Escreva "Nome do amigo", nome
      Escreva "Telefone do amigo", telefone
      Vapara Inicio
  FimSe
  Feche Agenda.dat
FimRotina
```

Note que foram introduzidos alguns elementos novos neste algoritmo, como por exemplo a declaração da estrutura do registro, que estamos colocando logo após o início da rotina principal. Esta declaração corresponde à instrução *struct* (discutida na página 49).
A instrução **Abra** AGENDA.DAT corresponde ao conjunto de instruções para a abertura do arquivo de dados.
Em seguida aparece o comando **Topo**, que faz com que o ponteiro de leitura seja posicionado no primeiro registro do arquivo (topo).
O rótulo [Inicio] indica o ponto de retorno para a leitura do arquivo. Em seguida é feita uma leitura no arquivo (**Leia Proximo registro** Agenda.data). Esta instrução obtém os dados de um registro apenas, que correspondem ao dados de um amigo.
Perguntamos em seguida se não é final de arquivo: **Se** nao FDA **entao**, se a resposta for afirmativa então devemos escrever os dados do amigo na tela e voltar para ler o próximo registro (**Vapara** Inicio), caso contrário, devemos fechar o arquivo (**Feche** Agenda.dat) e encerrar a rotina, pois o final do arquivo foi atingido.

Veja a seguir o algoritmo escrito em linguagem C :

```
/* Programa LEAGENDA.C */
#include<stdio.h>
#include<conio.h>
FILE *arquivo;
struct {
  int codigo;
  char nome[30];
  char telefone[15];
}registro;
main() {
   if((arquivo=fopen("AGENDA.DAT","r+b"))==NULL) {
      if((arquivo=fopen("AGENDA.DAT","wb"))==NULL) {
         printf("Erro de abertura");
         exit(1);
      }
   }
   fclose(arquivo);
   arquivo=fopen("AGENDA.DAT","r+b");
   clrscr();
   gotoxy(20,1);cprintf("Leitura do arquivo AGENDA.DAT");
```

```
   rewind(arquivo);
   inicio:
   if (fread(&registro, sizeof(registro), 1, arquivo) == 1) {
      gotoxy(10,3);cprintf("Codigo do amigo..: %d",registro.codigo);
      gotoxy(10,4);cprintf("Nome do amigo....: %-30s",registro.nome);
      gotoxy(10,5);cprintf("Telefone do amigo: %15s",registro.telefone);
      gotoxy(10,7);cprintf("Tecle algo para continuar ");
      getch();
      goto inicio;
   }
   fclose(arquivo);
}
```

Você não encontrará uma correspondência direta entre o algoritmo em Inter-S e o programa em linguagem C. Algumas instruções em Inter-S aparecem como várias linhas dentro do programa C, como é o caso da instrução **Abra** AGENDA.DAT, que no programa C aparece como:

```
if((arquivo=fopen("AGENDA.DAT","r+b"))==NULL) {
   if((arquivo=fopen("AGENDA.DAT","wb"))==NULL) {
      printf("Erro de abertura");
      exit(1);
   }
}
fclose(arquivo);
arquivo=fopen("AGENDA.DAT","r+b");
```

Vamos agora analisar as novas instruções utilizadas no programa LEAGENDA.C:

`rewind(arquivo);`

Esta instrução faz com que o ponteiro de leitura do arquivo seja posicionado no primeiro registro. Corresponde à *Topo* do Inter-S.

`if (fread(&registro, sizeof(registro), 1, arquivo) == 1) {`

Esta instrução lê um registro e ao mesmo tempo pergunta se não é final de arquivo. A função responsável pela leitura é a *fread*, que, entre parênteses, leva os seguintes argumentos:

`&registro` ..... Nome da variável (estrutura) que receberá os dados lidos
`sizeof(registro)` ..... Tamanho do registro que será lido
`1` ..... Quantidade de registros a serem lidos
`arquivo` ..... Nome do ponteiro que identifica o arquivo a ser lido

Observe que a função *fread* é comparada com o valor 1 (==1). Se fread for igual a 1 significa que não é final de arquivo.

Esta linha de programa corresponde às seguintes linhas do algoritmo Inter-S:

Leia Proximo registro Agenda.dat
Se não FDA **entao**

`gotoxy(10,4);cprintf("Nome do amigo....: %-30s",registro.nome);`

É claro que esta linha de comando não é mais segredo para você, com exceção do controle "%-30s" colocado na função *cprintf*.

Na realidade o que estamos fazendo é posicionando o cursor na coluna 10 da linha 4 (*gotoxy(10,4)*) e escrevendo, a partir dessa posição, a expressão "**Nome do amigo...:**" e em seguida o conteúdo do campo *registro.nome*, que corresponde ao nome do amigo que se encontra no registro lido. O controle "%-30s" faz com que o nome, que possui 30 caracteres, seja escrito da esquerda para a direita. Este alinhamento é obtido graças ao sinal de menos (-) colocado na frente do tamanho do campo.

Se você omitir este sinal, ou seja, se o controle for "%30s", o alinhamento do nome será feito da direita para a esquerda.

IMPORTANTÍSSIMO: *Digite o programa LEAGENDA.C, cujo código encontra-se a partir da página 55, e veja o seu funcionamento. Você terá agora dois programas: AGENDA.C, que permite a gravação dos dados de seus amigos, e LEAGENDA.C, que possibilita a leitura do arquivo.*

Atividades complementares

Sofistique os programas AGENDA.C e LEAGENDA.C, colocando cores e molduras. Torne-os mais agradáveis quanto à estética.

Apêndice A - Funções diversas

Neste apêndice você encontrará algumas funções de bibliotecas interessantes para deixar seus programas mais sofisticados. Mostramos o nome da biblioteca (aquela que deverá ser indicada na diretiva **`#include`**), a sintaxe requerida pela função e o seu propósito.

Entrada de caracter individual: getchar()

Biblioteca: stdio.h
Sintaxe....: `getchar();`
Propósito: A função getchar() (*get character*) lê um caracter individual da entrada padrão (em geral, o teclado).

Esta função é dita *line buffered*, isto é, não retorna valores até que o caracter de controle *line feed* (\n) seja lido. Este caracter, normalmente, é enviado pelo teclado quando a tecla [*enter*] é pressionada. Se forem digitados vários caracteres, estes ficarão armazenados no *buffer* de entrada até que a tecla [*enter*] seja pressionada. Então, cada chamada da função getchar() lerá um caracter armazenado no *buffer*.

Saída de caracter individual: putchar()

Biblioteca: stdio.h
Sintaxe....: `putchar(int c);`
Propósito: Esta função putchar() (*put character*) imprime um caracter individual *c* na saída padrão (em geral o monitor de vídeo).

Leitura de teclado: getch(), getche()

Biblioteca: conio.h
Sintaxe....:
```
getch();
getche();
```

Propósito: Estas funções fazem a leitura dos códigos de teclado. Estes códigos podem representar teclas de caracteres (A, y, *, 8, etc.), teclas de comandos ([enter], [delete], [Page Up], [F1], etc.) ou combinação de teclas ([Alt] + [A], [Shift] + [F1], [Ctrl] + [Page Down], etc.).

Ao ser executada, a função getch() (*get character*) aguarda que uma tecla (ou combinação de teclas) seja pressionada, recebe do teclado o código correspondente e retorna este valor. A função getche() (*get character and echoe*) também escreve na tela, quando possível, o caracter correspondente.

Código ASCII: ao ser pressionada uma tecla correspondente a um caracter ASCII, o teclado envia um código ao '*buffer*' de entrada do computador e este código é lido. Por exemplo, se a tecla A for pressionada o código 65 será armazenado no *buffer* e lido pela função.

Código Especial: ao serem pressionadas certas teclas (ou combinação de teclas) que não correspondem a um caracter ASCII, o teclado envia ao '*buffer*' do computador **dois** códigos, sendo o primeiro **sempre** 0. Por exemplo, se a tecla [F1] for pressionada os valores 0 e 59 serão armazenados e a função deve ser chamada **duas vezes** para ler os dois códigos.

Saída sonora: sound(), delay(), nosound()

Biblioteca: dos.h
Sintaxe....: `sound(freq);`
`delay(tempo);`
`nosound();`

Propósito: A função sound() ativa o alto-falante do PC com uma freqüência *freq* (Hz). A função delay() realiza uma pausa (aguarda intervalo de tempo) de duração *tempo* (milisegundos). A função nosound() desativa o alto-falante.

Redimencionamento de janela: window()

Biblioteca: *conio.h*
Sintaxe....: `window(esq, sup, dir, inf);`

Propósito: Esta função permite redefinir a janela de texto. As coordenadas *esq* e *sup* definem o canto superior esquerdo da nova janela, enquanto as coordenadas *inf* e *dir* definem o canto inferior direito da nova janela. Para reativar a janela padrão escreve-se a instrução **window(1,1,80,25)**. Quando uma janela é definida, o texto que ficar fora da janela fica congelado até que se redefina a janela original.

Monitoração de teclado: kbhit()

Biblioteca: *conio.h*
Sintaxe....: `kbhit();`

Propósito: Esta função *(keyboard hitting*) permite verificar se uma tecla foi pressionada ou não. Esta função verifica se existe algum código no buffer de teclado. Se houver algum valor, ela retorna um número não nulo e o valor armazenado no buffer pode ser lido com as funções getch() ou getche(). Caso nenhuma tecla seja pressionada a função retorna 0. Observe que, ao contrário de getch(), esta função **não aguarda** que uma tecla seja pressionada.

APÊNDICE B - TABELA ASCII

As tabelas mostradas neste apêndice representam os 256 códigos usados nos computadores da família IBM. Esta tabela refere-se ao *American Standard Code for Information Interchange* (código padrão americano para troca de informações), que é um conjunto de números representando caracteres ou instruções de controle usados para troca de informações entre computadores entre si, entre periféricos (teclado, monitor, impressora) e outros dispositivos. Estes códigos tem tamanho de 1 byte com valores de 00h a FFh (0 a 255 decimal). Podemos dividir estes códigos em três conjuntos: controle, padrão e estendido.

Os primeiros 32 códigos de 00h até 1Fh (0 a 31 decimal), formam o **conjunto de controle** ASCII. Estes códigos são usados para controlar dispositivos, por exemplo uma impressora ou o monitor de vídeo. O código 0Ch (*form feed*) recebido por ima impressora gera um avanço de uma página. O código 0Dh (*carriage return*) é enviado pelo teclado quando a tecla ENTER é pressionada. Embora exista um padrão, alguns poucos dispositivos tratam diferentemente estes códigos e é necessário consultar o manual para saber exatamente como o equipamento lida com o código. Em alguns casos o código também pode representar um caracter imprimível. Por exemplo o código 01h representa o caracter ☺ (*happy face*).

Os 96 códigos seguintes de 20h a 7Fh (32 a 127 decimal) formam o **conjunto padrão** ASCII. Todos os computadores lidam da mesma forma com estes códigos. Eles representam os caracteres usados na manipulação de textos: códigos-fonte, documentos, mensagens de correio eletrônico, etc. São constituídos das letras do alfabeto latino (minúsculo e maiúsculo) e alguns símbolos usuais.

Os restantes 128 códigos de 80h até FFh (128 a 255 decimal) formam o **conjunto estendido** ASCII. Estes códigos também representam caracteres imprimíveis porem cada fabricante decide como e quais símbolos usar. Nesta parte do código estão definidas os caracteres especiais: é, ç, ã, ü ...

Dec.	Hex.	Controle
0	00h	NUL (*Null*)
1	01h	SOH (*Start of Heading*)
2	02h	STX (*Start of Text*)
3	03h	ETX (*End of Text*)
4	04h	EOT (*End of Transmision*)
5	05h	ENQ (*Enquiry*)
6	06h	ACK (*Acknowledge*)
7	07h	BEL (*Bell*)
8	08h	BS (*Backspace*)
9	09h	HT (*Horizontal Tab*)
10	0Ah	LF (*Line Feed*)
11	0Bh	VT (*Vertical Tab*)
12	0Ch	FF (*Form Feed*)
13	0Dh	CR (*Carriage Return*)
14	0Eh	SO (*Shift Out*)
15	0Fh	SI (*Shift In*)
16	10h	DLE (*Data Link Escape*)
17	11h	DC1 (*Device control 1*)

18	12h	DC2 (*Device control 2*)
19	13h	DC3 (*Device control 3*)
20	14h	DC4 (*Device control 4*)
21	15h	NAK (*Negative Acknowledge*)
22	16h	SYN (*Synchronous Idle*)
23	17h	ETB (*End Transmission Block*)
24	18h	CAN (*Cancel*)
25	19h	EM (*End of Media*)
26	1Ah	SUB (*Substitute*)
27	1Bh	ESC (*Escape*)
28	1Ch	FS (*File Separator*)
29	1Dh	GS (*Group Separator*)
30	1Eh	RS (*Record Separator*)
31	1Fh	US (*Unit Separator*)

Binário	dec.	chr
00000000	0	
00000001	1	
00000010	2	
00000011	3	
00000100	4	
00000101	5	
00000110	6	
00000111	7	
00001000	8	
00001001	9	
00001010	10	
00001011	11	
00001100	12	
00001101	13	
00001110	14	
00001111	15	
00010000	16	
00010001	17	
00010010	18	
00010011	19	
00010100	20	
00010101	21	
00010110	22	
00010111	23	
00011000	24	
00011001	25	
00011010	26	
00011011	27	
00011100	28	
00011101	29	
00011110	30	
00011111	31	
00100000	32	spc
00100001	33	!
00100010	34	“
00100011	35	#
00100100	36	$
00100101	37	%
00100110	38	&
00100111	39	‘
00101000	40	(
00101001	41	)
00101010	42	*
00101011	43	+
00101100	44	,
00101101	45	-
00101110	46	.
00101111	47	/
00110000	48	0
00110001	49	1
00110010	50	2
00110011	51	3
00110100	52	4
00110101	53	5
00110110	54	6
00110111	55	7
00111000	56	8
00111001	57	9
00111010	58	:
00111011	59	;
00111100	60	<
00111101	61	=
00111110	62	>
00111111	63	?

Binário	dec.	chr
01000000	64	@
01000001	65	A
01000010	66	B
01000011	67	C
01000100	68	D
01000101	69	E
01000110	70	F
01000111	71	G
01001000	72	H
01001001	73	I
01001010	74	J
01001011	75	K
01001100	76	L
01001101	77	M
01001110	78	N
01001111	79	O
01010000	80	P
01010001	81	Q
01010010	82	R
01010011	83	S
01010100	84	T
01010101	85	U
01010110	86	V
01010111	87	W
01011000	88	X
01011001	89	Y
01011010	90	Z
01011011	91	[
01011100	92	\
01011101	93	]
01011110	94	^
01011111	95	_
01100000	96	`
01100001	97	a
01100010	98	b
01100011	99	c
01100100	100	d
01100101	101	e
01100110	102	f
01100111	103	g
01101000	104	h
01101001	105	i
01101010	105	j
01101011	107	k
01101100	108	l
01101101	109	m
01101110	110	n
01101111	111	o
01110000	112	p
01110001	113	q
01110010	114	r
01110011	115	s
01110100	116	t
01110101	117	u
01110110	118	v
01110111	119	w
01111000	120	x
01111001	121	y
01111010	122	z
01111011	123	{
01111100	124	|
01111101	125	}
01111110	126	~
01111111	127	

Binário	dec.	chr
10000000	128	Ç
10000001	129	ü
10000010	130	é
10000011	131	â
10000100	132	ä
10000101	133	à
10000110	134	å
10000111	135	ç
10001000	136	ê
10001001	137	ë
10001010	138	è
10001011	139	ï
10001100	140	î
10001101	141	ì
10001110	142	Ä
10001111	143	Å
10010000	144	É
10010001	145	æ
10010010	146	Æ
10010011	147	ô
10010100	148	ö
10010101	149	ò
10010110	150	û
10010111	151	ù
10011000	152	ÿ
10011001	153	Ö
10011010	154	Ü
10011011	155	¢
10011100	156	£
10011101	157	¥
10011110	158	₱
10011111	159	*f*
10100000	160	á
10100001	161	í
10100010	162	ó
10100011	163	ú
10100100	164	ñ
10100101	165	Ñ
10100110	166	ª
10100111	167	º
10101000	168	¿
10101001	169	_
10101010	170	¬
10101011	171	½
10101100	172	¼
10101101	173	¡
10101110	174	«
10101111	175	»
10110000	176	_
10110001	177	_
10110010	178	_
10110011	179	¦
10110100	180	¦
10110101	181	¦
10110110	182	¦
10110111	183	+
10111000	184	+
10111001	185	¦
10111010	186	¦
10111011	187	+
10111100	188	+
10111101	189	+
10111110	190	¬
10111111	191	+

Binário	dec.	chr
11000000	192	+
11000001	193	-
11000010	194	-
11000011	195	+
11000100	196	-
11000101	197	+
11000110	198	¦
11000111	199	¦
11001000	200	+
11001001	201	+
11001010	202	-
11001011	203	-
11001100	204	¦
11001101	205	-
11001110	206	+
11001111	207	-
11010000	208	-
11010001	209	-
11010010	210	-
11010011	211	+
11010100	212	+
11010101	213	+
11010110	214	+
11010111	215	+
11011000	216	+
11011001	217	+
11011010	218	+
11011011	219	_
11011100	220	_
11011101	221	¦
11011110	222	_
11011111	223	_
11100000	224	_
11100001	225	ß
11100010	226	_
11100011	227	¶
11100100	228	_
11100101	229	_
11100110	230	µ
11100111	231	_
11101000	232	_
11101001	233	_
11101010	234	_
11101011	235	_
11101100	236	_
11101101	237	_
11101110	238	_
11101111	239	_
11110000	240	_
11110001	241	±
11110010	242	_
11110011	243	_
11110100	244	_
11110101	245	_
11110110	246	÷
11110111	247	_
11111000	248	°
11111001	249	•
11111010	250	·
11111011	251	_
11111100	252	n
11111101	253	²
11111110	254	_
11111111	255	_