

Centro Universitário Feevale
Instituto de Ciências Exatas e Tecnológicas

A linguagem de programação 'C'
com o 'Turbo-c lite'

- Apostila -

por
Leandro Krug Wives

Julho de 2004

SUMÁRIO

1 HISTÓRICO.....	3
2 COMPOSIÇÃO DE UM PROGRAMA ESCRITO EM C	4
3 TIPOS DE DADOS.....	5
3.1 Declarando variáveis.....	7
4 PRINCIPAIS OPERADORES DA LINGUAGEM C:	8
5 FUNÇÕES DE ENTRADA E SAÍDA DE DADOS:	9
5.1 função printf().....	9
5.2 função scanf()	11
6 COMANDOS DE REPETIÇÃO	12
6.1 O comando “while”	12
6.2 O comando “do”	13
6.3 O comando “for”	13
7 COMANDOS DE DESVIO CONDICIONAL / TOMADA DE DECISÃO	15
7.1 O comando “if”	15
7.2 O comando “switch”	16
8 TRABALHANDO COM ESTRUTURAS	18
9 TRABALHANDO COM VETORES/ARRAYS (E STRINGS).....	20
10 TRABALHANDO COM MATRIZES	22
11 TRABALHANDO COM PONTEIROS.....	23
11.1 Aritmética de ponteiros	24
12 ALOCAÇÃO DINÂMICA DE MEMÓRIA	25
13 SOBRE FUNÇÕES.....	29
13.1 Definindo funções.....	31
13.2 Escopo de variáveis: globais e locais.....	33
13.3 Considerações sobre a passagem de parâmetros	35
14 APÊNDICE I: MANUAL DO TURBO-C LITE (TCLITE).....	38
14.1 O ambiente tclite.....	38
14.2 O menu	38
14.2.1 O menu do sistema (-).....	39
14.2.2 O menu de arquivos (File) [ALT + F]	39
14.2.3 Menu de edição (Edit)	40
14.2.4 Janela de Busca (Search)	40
14.2.5 Menu de execução (Run)	41
14.2.6 Menu de compilação (Compile).....	41
14.2.7 Menu de depuração (Debug).....	41
14.2.8 Menu de projeto (Project)	42
14.2.9 Menu de opções (Options)	42
14.2.10 Menu de Janela (Window)	43
14.2.11 Menu de Ajuda (Help)	43
14.3 A janela de edição.....	44
14.4 A janela de mensagens e depuração de variáveis	44
15 APÊNDICE II: USANDO O DEPURADOR DO TURBO-C.....	45
15.1 Verificando o conteúdo de variáveis.....	46
15.2 Adicionando pontos de parada (breakpoints)	47
16 APÊNDICE III: TABELA ASCII.....	48
16.1 Programa exemplo que imprime a tabela ASCII na tela do seu computador	50
17 APÊNDICE IV: RESUMO GERAL.....	51
18 LEITURA RECOMENDADA	52

1 HISTÓRICO

A linguagem C foi desenvolvida na década de 70 nos laboratórios da Bell Computers. Brian Kernighan e Dennis Ritchie definiram essa linguagem (sua sintaxe, na verdade) com o objetivo de facilitar a construção do sistema operacional UNIX, seus comandos e utilitários. Isso porque os sistemas operacionais eram feitos em linguagem Assembly, tornando sua construção e, principalmente, portabilidade difícil. A grande maioria dos programas para o ambiente UNIX escritos até o momento utilizam essa linguagem.

Durante algum tempo, após a definição de Kernighan e Ritchie, vários laboratórios e universidades norte-americanos desenvolveram a sua versão da linguagem C. Essas versões tinham suas peculiaridades e não eram compatíveis entre si.

Para solucionar esse problema, o American National Standards Institute – ANSI¹ (Instituto Nacional de Padrões Americano) criou em 1983 um comitê encarregado de definir uma linguagem C não ambígua e independente de máquina, dando origem ao padrão ANSI-C.

A princípio, qualquer programa pode ser escrito em linguagem C. Se forem seguidos os padrões ANSI, um programa C pode ser compilado e executado em qualquer computador que aceite a linguagem C.

Essa linguagem é muito poderosa, fornecendo acesso ao nível de máquina e contendo muitos operadores que outras linguagens não tem. Por esse motivo ela costuma ser a linguagem preferida para a construção de sistemas operacionais (como o LINUX, por exemplo), jogos de computador, utilitários de manipulação de disco e memória, antivírus, entre outros.

¹ <http://www.ansi.org>.

2 COMPOSIÇÃO DE UM PROGRAMA ESCRITO EM C

Um programa em C constitui-se basicamente de funções e variáveis. As funções contêm os comandos que especificam as operações de computação a serem feitas. As variáveis armazenam os valores utilizados durante a computação. As funções são sub-rotinas similares aos procedimentos (*procedures*) do Pascal.

Um programa fonte C pode ser dividido nas seguintes partes:

- Zona de comentários (descrição do programa);
- Zona de inclusões (declaração das bibliotecas utilizadas no programa);
- Zona de definições (declaração de constantes e/ou variáveis globais);
- Zona de funções do programa.

Zona de comentários	<pre> /* ***** Programa: exemplol.c Autor : Leandro Krug Wives Objetivo: Demonstrar as regiões básicas de um programa escrito na linguagem C ***** */ </pre>
Zona de inclusões	<pre> #include <stdio.h> #include <conio.h> </pre>
Zona de definições	<pre> #define DIAS_DO_ANO 360 #define DIAS_DO_MÊS 30 </pre>
Zona de funções do programa	<pre> Definição dos argumentos do programa principal int main(void) { Definição de variáveis utilizadas no programa principal int idade_jose; int numero_dias; int numero_meses; idade_jose = 22 ; Comandos de processamento (algoritmo) do programa principal numero_dias = idade_jose * DIAS_DO_ANO; numero_meses = numero_dias / DIAS_DO_MES; printf("Jose já viveu %d anos, %d meses e %d dias", idade_Jose, numero_meses, numero_dias); getch(); Finalização do programa principal return(0); } </pre>

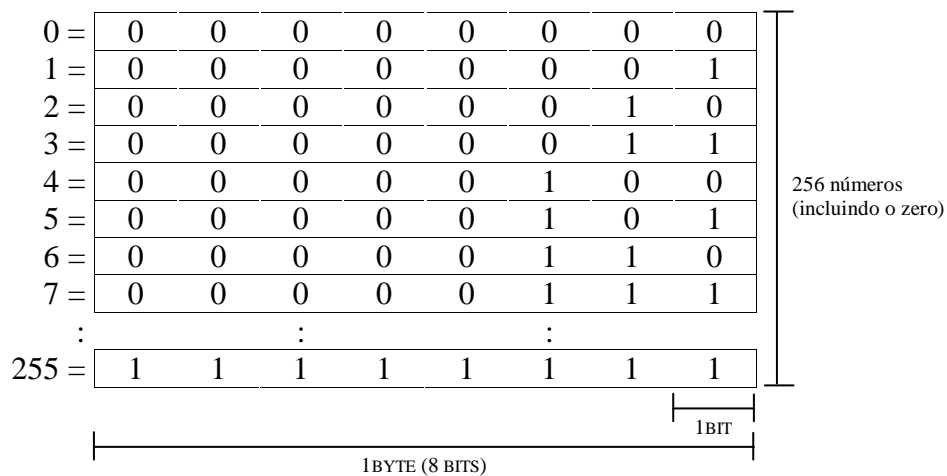
Um programa escrito na linguagem C é constituído de uma ou mais funções. Independente do número de funções que ele possuir, a execução é sempre iniciada em uma função especial denominada *main* (que significa *principal* em inglês). Todo programa deve possuir uma e somente uma função principal (*main*). Se ela não for declarada o compilador gera um erro, pois sem ela o programa não tem como começar. O exemplo acima só utiliza a função *main*.

3 TIPOS DE DADOS

A linguagem C possui os seguintes tipos de dados básicos: *char*, *int*, *float* e *double*.

O tipo *char* ocupa 1 byte de memória. Cada byte possui 8 bits. Os bits são as unidades básicas de armazenamento. Um bit pode estar ligado (1) ou desligado (0), logo, cada bit pode conter o valor 0 ou 1.

A memória do computador pode ser vista como uma seqüência finita de bits. As informações (números, letras) são armazenadas na memória como uma seqüência de bits. Cada número possui sua seqüência correspondente:



Com oito bits, sendo que cada bit pode conter dois valores (ligado ou desligado), é possível fazer 2^8 (256) combinações de zeros e uns. Isso significa que o número máximo que cada byte pode armazenar é 255.

O tipo *char* padrão utiliza, na verdade, somente 7 bits para armazenar um número ($2^7=128$). Um dos bits (chamado bit mais significativo) é utilizado para indicar se o número é positivo ou negativo. Com isso, o tipo *char* pode armazenar números entre -128 até +127 (256 combinações de bits).

Se o programador souber que seu programa vai trabalhar somente com números positivos, ele pode utilizar o bit de sinal para o armazenamento de números. Para tanto é necessário utilizar o modificador de tipo **unsigned**. Uma variável do tipo *unsigned char* pode armazenar, portanto, números entre 0 e 255. Essa quantidade (256) corresponde exatamente aos 256 caracteres da tabela ASCII. Logo, o tipo *char* é comumente utilizado para armazenar caracteres (letras).

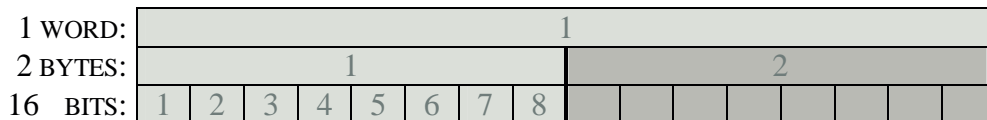
O tipo *int* ocupa 1 WORD (palavra). A *palavra* ou WORD é o nome dado a menor unidade de informação que um computador processa. Há alguns anos atrás os computadores possuíam processadores de 8 bits (1 byte). Com o tempo eles foram evoluindo, passando por 16 bits (2 bytes), 32 bits (4 bytes) e, atualmente, existem computadores que processam 64 bits (8 bytes).

O *int*, como o próprio nome diz, serve para armazenar números inteiros.

Caso o programador necessite armazenar números fracionários, ele deve utilizar o tipo *float* ou o tipo *double*. O *float* ocupa 2 WORDS e consegue armazenar números de ponto flutuante de precisão simples. O tipo *double* ocupa 4 WORDS e consegue armazenar números de ponto flutuante de precisão dupla.

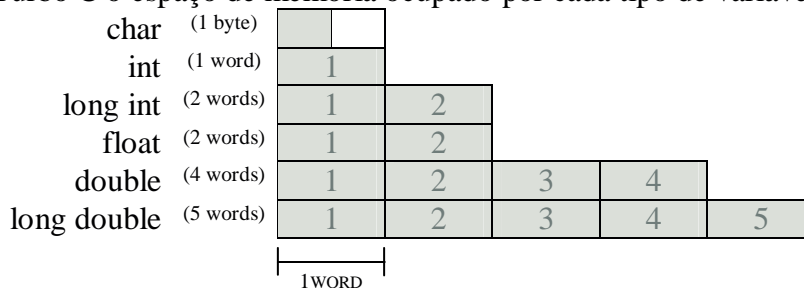
No ambiente de programação **Turbo C** da Borland, 1 WORD equivale a 2 BYTES (ou seja, ocupa o mesmo espaço que 2 bytes). Como cada BYTE equivale a 8 BITS, os 2 BYTES ocupam o espaço de 16 BITS. Logo, 1 WORD equivale a 16 BITS.

Veja:



$$1 \text{ WORD} = 2 \text{ BYTES} = 16 \text{ BITS}$$

No Turbo C o espaço de memória ocupado por cada tipo de variável é o seguinte:



Nesse ultimo quadro verifica-se que existem ainda outros tipos de variáveis no C: o *long int* e o *long double*. Há ainda o *short int*, mas esse é igual ao *int* no Turbo C.

Em resumo, os tipos de variáveis que você pode utilizar no C são os seguintes:

	int	inteiros entre -32.768 e +32.767
	unsigned int	inteiros entre 0 e 65.535
Números inteiros	long int	inteiros entre -2.147.483.648 e +2.147.483.648
	unsigned long int	inteiros entre 0 e 4.294.967.296
	short int	inteiros entre -32.768 e +32.767
números de ponto flutuante	float	ponto flutuante entre 3,4e-38 a 3,4e+38
	double	ponto flutuante entre 1,7e-308 a 1,7e+308
	long double	ponto flutuante entre 3,4e-4932 a 1,1e+4932
caracteres	char	caracteres ou números entre -128 a +127
	unsigned char	caracteres ou números entre 0 e 255

3.1 Declarando variáveis

Em 'C', para criar uma variável você indica o tipo seguido pelo nome da variável.

Exemplo:

int idade; → cria uma variável chamada *idade* que pode armazenar números inteiros (entre -32.768 e +32.767).

A princípio você pode dar qualquer nome a uma variável, desde que esse nome não corresponda a uma das **palavras reservadas**² do C. Outra restrição é a de que os nomes devem começar com uma letra ou um sublinhado e não podem conter símbolos tais como \$ % ^ #. A partir da segunda letra, números também podem ser utilizados.

Exemplos de nomes de variáveis válidos:

```
int _idade;
char _i2;
int i2;
float __ab2b_ad;
int _2b;
```

Exemplos de nomes de variáveis inválidos:

```
int 2a;
char %ab;
int c$as;
```

Outra observação é a de que o 'C' faz distinção entre caracteres maiúsculos e minúsculos. Isso significa que *Idade* é diferente de *idade*, *IDADE* e *IdAdE*. Sendo assim, você pode criar variáveis cujo nome correspondam às palavras reservadas (desde que o utilizem letras maiúsculas), pois, por exemplo, *INT* é diferente de *int*.

Aconselha-se que os nomes não ultrapassem 32 caracteres (letras) pois o compilador reconhece somente os primeiros 32.

² As palavras reservadas do C são as seguintes: void, int, float, char, double, short, long, unsigned, signed, volatile, const, auto, static, extern, register, enum, struct, union, typedef, case, default, **sizeof**, **break**, **continue**, **return**, **do**, **if**, **else**, **for**, **goto**, **switch** e **while**. Pode-se notar que as palavras reservadas correspondem aos tipos básicos de variáveis do C, além de modificadores de tipos e comandos da linguagem (em negrito).

4 PRINCIPAIS OPERADORES DA LINGUAGEM C:

- Operador de atribuição (=): Serve para colocar (atribuir) um dado (um valor) em uma variável:

```
int  a; // declara uma variável que armazena números inteiros
char b; // declara uma variável que armazena caracteres
float c; // declara uma variável que armazena números reais
int  d; // declara uma variável que armazena números inteiros
int  e; // declara uma variável que armazena números inteiros

a = 10; // coloca o número 10 na variável a
b = 'a'; // coloca o caractere 'a' na variável b
c = 13.4; // coloca o número real 13.4 na variável c
d = a; // coloca o conteúdo de a (que é 10) em d
e = ( a + d ) * 3; // soma o conteúdo de a com o de d, multiplica o
// resultado por 3 e coloca-o na variável e
```

Note, pelos dois últimos exemplos, que você também pode utilizar variáveis e até mesmo expressões nas atribuições.

- Operadores matemáticos:

- Soma (+): a + b
- Subtração (-): a - b
- Multipliação (*): a * b
- Divisão (/): a / b
- Resto de divisão (%): a % b *(retorna o resto da divisão de a por b)*
- Incremento (++): a++ *(soma 1 ao conteúdo da variável)*
- Decremento (--): a-- *(diminui 1 do conteúdo da variável)*

- Operador de teste de igualdade (==): retorna verdadeiro se o que estiver do lado esquerdo do operador for igual a o que estiver do seu lado direito. Serve para testar se o conteúdo de uma variável é igual a determinado dado ou valor:

```
a == 1 (retorna verdadeiro se o conteúdo da variável a for igual a 1)
a == b (retorna verdadeiro se o conteúdo da variável a for igual ao da b)
```

- Operador de teste de diferença (!=): retorna verdadeiro se o que estiver do lado esquerdo do operador for diferente do que estiver do seu lado direito. Serve para testar se o conteúdo de uma variável é diferente de determinado dado ou valor:

```
a != 1 (retorna verdadeiro se o conteúdo de a for diferente de 1)
a != b (retorna verdadeiro se o conteúdo de a for diferente do de b)
```

- Operadores lógicos:

- Menor (<): a < b *(verdadeiro se o conteúdo de a < do que o de b)*
- Maior (>): a > b *(verdadeiro se o conteúdo de a > do que o de b)*
- E (&&): (a == 1) && (b == 10) *(verdadeiro se a == 1 E b == 10)*
- OU (||): (a == 1) || (b == 10) *(verdadeiro se a == 1 OU b == 10)*
- NEGAÇÃO (!): !(a == b) *(retorna verdadeiro se a for diferente de b)*

5 FUNÇÕES DE ENTRADA E SAÍDA DE DADOS:

5.1 função printf()

A função `printf` serve para que o programador possa colocar informações na tela do computador. O nome dela vem de *print formatted* (impressão formatada), o que significa que ela oferece opções de formação que permitem com que o programador escolha como um dado ou variável deve ser mostrado na tela.

Sintaxe (forma de utilização):

```
printf("expressão de controle", argumento_1, argumento_2, ..., argumento_n );
```

A *expressão de controle* é uma string que especifica o formato que cada argumento (que pode ser um valor, dado ou variável) deve ser mostrado na tela. Alguns formatos possíveis que podem ser utilizados na expressão de controle são os seguintes:

%d	imprime o argumento no formato número (inteiro decimal)
%c	imprime o argumento no formato caracter (letra)
%f	imprime o argumento no formato de ponto flutuante (float)
%s	imprime o argumento no formato cadeia de caracteres (string)

Os *argumentos* correspondem aos dados ou variáveis que o programador quer mostrar na tela.

Exemplos:

```
comando : printf("%d", 65); // imprime 65 como um número
resultado: 65
comando : printf("%c", 65); // imprime 65 como um caractere
resultado: A
comando : float media = 20.4;
          printf("%f", media); // imprime o conteúdo de média como um float
resultado: 20.4
comando : float media = 20.4;
          printf("%d", media); // imprime o conteúdo de média como um int
resultado: 20
comando:  printf("%s", "olá, como vai?");
resultado: olá, como vai?
comando:  printf("olá, como vai?");
resultado: olá, como vai?
```

Note, no último exemplo, que as strings podem ser colocadas diretamente na expressão de controle. Lembre-se, porém, que isso só funciona com strings!

Na verdade, a expressão de controle pode ser mais complexa, e tem a seguinte forma (os itens entre colchetes são opcionais):

`%[-][tamanho][.precisão]tipo`

O símbolo `%` é denominado código de controle. O código de controle, além de especificar o formato de saída de um argumento ele também serve para especificar seus tamanhos e alinhamentos.

O `tipo` corresponde a um dos tipos da tabela anterior (`c`, `d`, `f`, `s`).

Os demais itens são opcionais, e significam o seguinte:

- → diz para o `printf` que o argumento deve ser alinhado à esquerda. Se ele for suprimido (não aparecer) o `printf` alinha pela direita.
- tamanho* → indica o espaço mínimo que o argumento deve ocupar na tela.
- precisão* → indica quantas casas após a vírgula devem ser utilizadas (quando o argumento for um número real do tipo `float` ou `double`).

Exemplos:

```
Comando : printf("012345678901234567890123456789\n");
          printf("%10s%10c%10s\n", Ano, ' ', Valor);
          printf("%9d%11c%10d\n", 1, ' ', 1000);
          printf("%9d%11c%10d\n", 2, ' ', 2500);
          printf("%9d%11c%10d\n", 3, ' ', 3800);
```

```
Resultado: 012345678901234567890123456789
           Ano          Valor
           1            1000
           2            2500
           3            3800
```

```
Comando : printf("012345678\n");
          printf("%4.2f\n", 0.8);
          printf("%8.5f\n", 0.8);
          printf("%-8.2f\n", 0.8);
```

```
Resultado: 012345678
           0.80
           0.80000
           0.80
```

5.2 função scanf()

Essa função serve para que o programador pedir informações ao usuário e colocá-las em variáveis. O nome vem de *scan formatted* (algo como leitura formatada), o quê significa que o programador deve dizer em que formato o dado que o usuário digitar deve ser colocado na variável.

Sintaxe:

```
scanf("expressão de controle", endereço_1, endereço_2, ..., endereço_n);
```

A *expressão de controle*, como no caso anterior, serve para indicar o formato como o dado digitado pelo usuário deve ser colocado em uma variável. Os *endereços* correspondem às posições de memória das variáveis que vão receber as informações.

A princípio a scanf não tem como saber o tipo de variável que o usuário está passando para ela como parâmetro. Logo, é pela *expressão de controle* que ela sabe disso. Portanto, você deve colocar o tipo correspondente à variável que está passando como parâmetro.

Os tipos são os mesmos utilizados no printf:

%d	lê um argumento no formato número (inteiro decimal)
%c	lê um argumento no formato caracter (letra)
%f	lê um argumento no formato de ponto flutuante (float)
%s	lê um argumento no formato cadeia de caracteres (string)

Para passar o endereço de uma variável, basta utilizar o operador **&**, que retorna o endereço de uma variável.

Exemplos:

```
int idade;
printf("Digite sua idade: ");
scanf("%d", &idade); // lê um número e coloca-o na variável idade
printf("Puxa, você tem %d anos!\n", idade);
char nome[30];
printf("Olá, qual é o seu nome? ");
scanf("%s", nome); // lê um string e coloca-o na variável nome
printf("%s, que nome bonito!", nome);
```

Note que no último caso não foi necessário passar o endereço da variável nome. Na verdade, nome é um vetor de caracteres (`char nome[30]`) e, nesses casos, a própria variável já contém o endereço do primeiro elemento do vetor. Lembre-se de nunca colocar o **&** na frente de strings e sempre coloca-lo nas outras variáveis!

6 COMANDOS DE REPETIÇÃO

Esses comandos devem utilizados quando o programador necessita implementar um trecho de programa que deve ser repetido até que determinada condição seja satisfeita. Enquanto a condição não for satisfeita, o laço de execução deve ser repetido. A linguagem C possui três comandos básicos de repetição: o comando *while* (enquanto), o comando *do* (faça) e o comando *for* (para).

6.1 O comando “while”

Quando o programa em execução encontra o comando *while* (enquanto) ele primeiro avalia se a *condição* é verdadeira. Se ela for verdadeira, o trecho de código que está dentro do laço (dentro das chaves) é executado. Após a execução, o programa volta para a linha do *while* para testar se a condição ainda é verdadeira. Se isso ocorrer, o trecho é executado novamente. Esse laço (loop) ocorre até que a condição se torne falsa.

Sintaxe:

```
while(condição)
{
    // trecho de código a ser repetido enquanto a condição é verdadeira
}
```

Lê-se:

```
enquanto (condição for verdadeira)
{
    // execute esses comandos
}
```

OBS: Tome cuidado. Se o trecho de código que está dentro do laço não for capaz de alterar a condição o laço pode ser executado para sempre!

Exemplo:

```
#include <stdio.h>
void main(void)
{
    int contador = 0;           // inicializa o contador em zero
    while(contador < 10)      // enquanto o contador for menor do que 10
    {
        printf("%d\n", contador); // imprime o contador
        contador++;             // incrementa o contador
    }
}
```

6.2 O comando “do”

O comando *do* (faça) também serve para criar laços de repetição dentro de um programa. Porém, ao contrário do comando *while*, onde os comandos só são executados se a condição for verdadeira (ou seja, primeiro a condição é testada e depois os comandos são executados), primeiro os comandos são executados e depois a condição é testada. Com isso, o trecho de código é executado ao menos uma vez.

Sintaxe:

```
do
{
    // trecho de código a ser repetido
}while(condição)
```

Lê-se:

```
faça
{
    // execução desses comandos
}enquanto (condição for verdadeira);
```

OBS: O do sempre é acompanhado do while (ao final).

Exemplo:

```
#include <stdio.h>
void main(void)
{
    int contador = 0;           // inicializa o contador em zero

    do{                         // faça:
        printf("%d\n", contador); // imprime o contador
        contador++;            // incrementa o contador
    }while(contador < 10)      // enquanto o contador for menor do que 10

}
```

6.3 O comando “for”

For (para) também é um comando de repetição onde a condição é testada antes do trecho ser executado. Ele é portanto muito parecido com o comando *while*, porém, como em outras linguagens, ele é geralmente utilizado para executar o código durante um número específico de vezes.

Sintaxe:

```
for(inicialização; condição; incremento_ou_decremento)
{
    // trecho de código a ser repetido
}
```

Lê-se:

```
para (a iniciando em zero; enquanto a < 10; incremente o valor de a)
{
    // execute esses comandos
}
```

Exemplo:

```
for(a = 0; a < 10; a = a + 1)
{
    printf("%d\n", a);
}
```

Esse exemplo é similar ao seguinte, que utiliza o comando *while*:

```
a = 0; // inicia a em zero
while(a < 10) // enquanto a menor do que 10
{
    printf("%d\n", a);
    a = a + 1; // e aumente o valor de a em 1
}
```

Note que no comando *while* a inicialização da variável deve ser colocada em qualquer posição anterior ao comando. Da mesma forma, o programador deve colocar o incremento (ou decremento) da variável de controle dentro do laço (dentro das chaves), pois o comando *while* não faz isso automaticamente. Essas duas restrições devem ser seguidas corretamente para que o programa funcione. Mesmo que o programador esqueça de segui-las (esqueça de colocar a inicialização e o incremento) o compilador C vai gerar o programa e executa-lo. É obrigação do programador saber se ele inicializou a variável e se ele a incrementou. Em programas muito extensos pode ser complicado controlar esse tipo de coisa.

O comando *for* facilita o controle desse tipo de problema, pois permite juntar a inicialização, a condição e o incremento (ou decremento) da variável de controle em uma única linha, facilitando sua visualização e mudança (caso necessário). Ao contrário de outras linguagens, que automaticamente incrementam ou decrementam a variável de controle de laço, no C é necessário que o programador faça isso. Com isso, o programador torna-se livre para fazê-lo da maneira que for de seu interesse, como por exemplo, aumentar o valor da variável em 2.

Exemplos:

```
for(a = 0; a < 10; a = a + 2); // incrementa a em 2
for(a = 0; a < 10; a = a + 4); // incrementa a em 4
for(a = 10; a > 0; a = a - 1); // decrementa a em 1
for(a = 10; a > 0; a = a - 4); // decrementa a em 4
```

7 COMANDOS DE DESVIO CONDICIONAL / TOMADA DE DECISÃO

7.1 O comando “if”

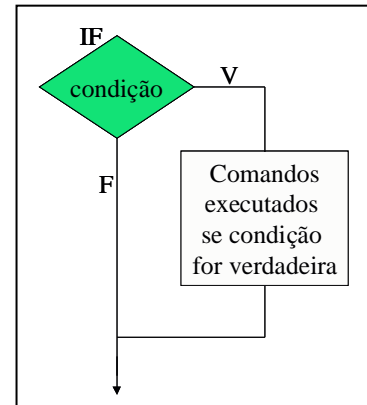
Esse comando é utilizado quando o programador tem um comando ou bloco de comando a ser executado somente se uma condição for verdadeira. Ou seja, se o usuário fez alguma coisa, os comandos são executados; se ele não fez, eles não são executados.

Sintaxe:

```
if(condição)
{
    // trecho de código a ser executado se
    // a condição for verdadeira
}
```

Lê-se:

```
se(condição for verdadeira)
{
    // execute esses comandos
}
```



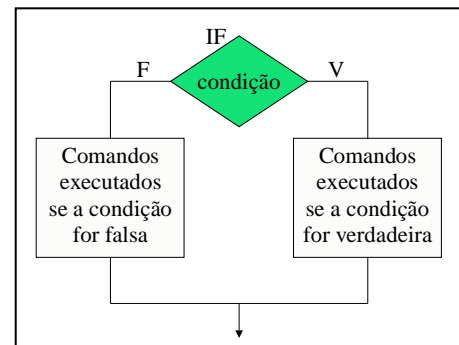
O comando *if* pode vir acompanhado do comando *else* (senão). Eles são utilizados quando o programador tem dois comandos ou blocos de comandos mas deseja que somente um deles seja executado. A escolha de qual deles vai ser executado depende de uma condição do programa, que pode ser modificada de acordo, por exemplo, com uma escolha que usuário fez.

Sintaxe:

```
if(condição)
{
    // trecho de código a ser executado se condição for verdadeira
}
else
{
    // trecho de código a ser executado se condição for falsa
}
```

Lê-se:

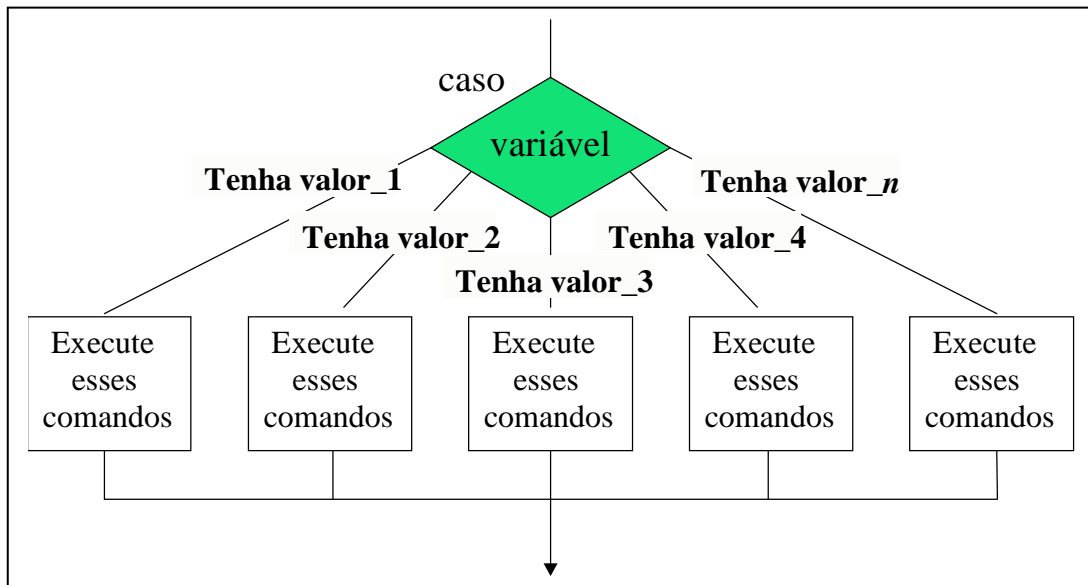
```
se(condição for verdadeira)
{
    // execute esses comandos
}
senão (ou caso contrário)
{
    // execute esses outros comandos
}
```



OBS: Nesse caso, somente um dos dois blocos de comandos é executado. Nunca os dois são executados.

7.2 O comando “switch”

O comando *if* funciona muito bem quando o número de blocos é igual a dois, ou seja, deve-se escolher se um ou outro vai ser executado. Porém, quando há uma quantidade bem maior de códigos, a utilização do *if* pode ser bem complexa, necessitando aninhá-los (coloca-los uns dentro de outros). Para que isso não ocorra, pode-se utilizar o comando *switch* (troca ou escolha).



Sintaxe:

```
switch(variável ou condição)
{
  case valor_1: // bloco de código
                break;
  case valor_2: // bloco de código
                break;
  case valor_3: // bloco de código
                break;
  case valor_4: // bloco de código
                break;
  :
  case valor_n: // bloco de código
                break;
  default:     // bloco de código padrão caso nenhum dos outros
                // valores seja encontrado
}
}
```

Esse código pode ser compreendido da seguinte forma:

```
De acordo com o conteúdo da (variável ou condição)
{
  caso seu conteúdo seja valor_1: execute esse código
                                  pare;
  caso seu conteúdo seja valor_2: execute esse código
                                  pare;
  caso seu conteúdo seja valor_3: execute esse código
                                  pare;
  caso seu conteúdo seja valor_4: execute esse código
                                  pare;
  :
  caso seu conteúdo seja valor_n: execute esse código
                                  pare;
  caso não seja nenhum dos anteriores: execute esse código padrão
}
}
```

Assim, de acordo com a condição um dos diversos blocos definidos vai ser executado de acordo.

Note que é necessário que o programador coloque o comando break (pare) no final de cada bloco de código. Se o break não for encontrado, o próximo bloco de código também é executado. Em alguns casos essa característica é extremamente útil. Veja o seguinte exemplo:

```
switch(tecla)
{
    case 'a': // segue para o próximo case
    case 'A': printf("você digitou a tecla A");
               break;
    case 'b': // segue para o próximo case
    case 'B': printf("você digitou a tecla B");
               break;
}
```

Neste caso, o printf("você digitou a tecla A") é executado quando tecla vale 'a' e quando tecla vale 'A'.

8 TRABALHANDO COM ESTRUTURAS

Estrutura é uma facilidade que permite agrupar vários dados/variáveis (que possuam alguma relação) em um único identificador. Isso não só facilita a organização dos dados como também a sua manipulação (envio e recebimento de dados em funções).

Com uma estrutura o programador consegue representar de forma mais natural algum objeto do mundo real. Depois de um objeto do mundo real ter sido modelado, ou seja, após seus atributos importantes (e relevantes) terem sido identificados, deve-se criar um tipo de dado que consiga representá-lo na memória do computador. Na linguagem C isso é feito por uma estrutura.

Para se criar uma estrutura na linguagem C usa-se o comando `struct`, cuja forma geral é:

```
struct <identificador_da_estrutura>
{
    <tipo_do_atributo> <identificador_do_primeiro_atributo>;
    <tipo_do_atributo> <identificador_do_segundo_atributo>;
    ⋮
    <tipo_do_atributo> <identificador_do_último_atributo>;
};
```

Exemplo:

```
Struct Pessoa
{
    char nome[30];
    int idade;
    char sexo;
};
```

No trecho de código acima é definido um novo tipo de variável chamada Pessoa. Esse novo tipo (que foi definido pelo programador) serve para armazenar na memória do computador informações sobre pessoas. Esse tipo de variável possui como atributos (sub-variáveis) o *nome*, a *idade* e o *sexo* de uma pessoa. Note que os atributos foram definidos através de tipos básicos do C: vetor/cadeia de caracteres (o tipo `char[]`), números inteiros (`int`) e caractere simples (`char`). Porém, estes atributos (variáveis) poderiam ter sido definidos por qualquer outro tipo, inclusive tipos complexos (outras estruturas como *Pessoa*) definidos previamente pelo programador.

A partir deste momento podem ser criadas variáveis na memória do computador capazes de armazenar informações sobre o objeto modelado:

```
Pessoa Funcionario;
```

Estas variáveis são manipuladas (acessadas) através da seguinte notação:

```
<objeto>.<atributo>
```

Ou seja, o ponto '.' é o caractere especial utilizado para indicar que se deseja manipular um atributo do objeto. Coloca-se o identificador do objeto, o ponto e o identificador do atributo. Estes atributos podem ser utilizados normalmente, como se fossem variáveis comuns, inclusive por procedimentos e funções de manipulação de variáveis oferecidas pela linguagem:

```
strcpy(Funcionario.nome, "Márcia");
Funcionario.idade = 24;
printf("%s", Funcionario.nome);
Funcionario.sexo = 'F';
scanf("%d", &Funcionario.idade);
```

Estas variáveis funcionam como se fossem variáveis normais, inclusive são perdidas quando o programa é finalizado. Para que os conteúdos destas variáveis sejam salvos, para que possam ser manipulados em outra execução do programa, é necessário armazenar o conteúdo destas variáveis (estruturas) em um arquivo.

9 TRABALHANDO COM VETORES/ARRAYS (E STRINGS)

Imagine que você tenha que solicitar para o usuário diversos objetos de um mesmo tipo, como, por exemplo, ler dez números. Normalmente, você criaria dez variáveis e, em seguida, atribuiria os valores a cada uma delas:

```
int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;  
n1 = 10; n2 = 30; ... n10 = 10;
```

Ou pediria para o usuário digitar os valores através de diversas chamadas à função `scanf`:

```
scanf("%d", &n1); scanf("%d", &n2); ... scanf("%d", &n10);
```

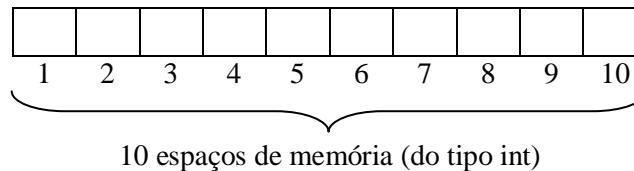
Essa seria uma solução simples, mas, em alguns casos, é necessário manipular diversos números (100, 1000 ou mais), e aí sim essa solução torna-se complicada.

Nestes casos, onde há diversos objetos de um mesmo tipo, é possível utilizar um vetor (*array* em inglês). Com um vetor, uma única variável armazena todos os objetos. A única coisa que varia é o índice, ou seja, a posição do objeto no vetor.

Em C os vetores são criados simplesmente colocando o número de elementos do vetor entre colchetes após o nome da variável:

```
int n[10]; // cria um vetor de 10 elementos de tipo int
```

Nesse caso, o C aloca 10 espaços contíguos de memória que serão utilizados para armazenar cada um dos elementos do vetor.



Cada elemento do vetor pode ser utilizado como se fosse uma variável independente. Para utilizar um elemento do vetor, basta indicar dentro dos colchetes o índice correspondente ao elemento desejado:

```
n[0] = 1; // atribui o valor 1 ao primeiro elemento do vetor  
n[5] = 23; // atribui o valor 23 ao sexto elemento do vetor  
n[9] = 33; // atribui o valor 33 ao último elemento do vetor
```

Note, no exemplo acima, que o primeiro elemento do vetor é o elemento de índice zero (0). Logo, um vetor de tamanho n , começa em zero e termina em $n - 1$ (ou seja, se o vetor foi declarado com 10 elementos, o décimo elemento é o elemento de número 9).

Índice:	0	1	2	3	4	5	6	7	8	9
Elemento:	1°	2°	3°	4°	5°	6°	7°	8°	9°	10°

Você pode criar vetores de qualquer tipo, inclusive de estruturas definidas pelo programador:

```
float h[20]; // cria um vetor de 20 elementos de tipo float
char p[15]; // cria um vetor de 15 elementos de tipo char
```

Os vetores do tipo *char* merecem uma atenção especial. Isso porque a linguagem C não possui o tipo de dados *string*. Logo, se o programador necessitar armazenar uma string em memória ele deve criar um vetor de caracteres (vetor de *chars*).

Apesar de não possuir o tipo string, o C oferece uma biblioteca com uma enorme variedade de funções que manipulam vetores de caracteres como se fossem strings. Para que essas funções funcionem corretamente, é necessário que os vetores de caracteres possuam sempre um símbolo especial que indica onde o vetor acaba. Esse símbolo é o '\0'. Ele é chamado de terminador de strings.

Com isso, sempre que um vetor de caracteres for criado e uma palavra for adicionada a ele, o caractere '\0' deve ser colocado após:

L	E	O	N	A	R	D	O	\0	
---	---	---	---	---	---	---	---	----	--

Na string anterior a palavra Leonardo é seguida do terminador (\0). Assim, todas as funções que manipulam strings sabem onde a palavra acaba. A função printf é uma destas funções que necessitam saber onde a string acaba. Assim, ela vai mostrando na tela todos os caracteres da string (que é um vetor de caracteres) até chegar em um símbolo \0. Se o terminador não existisse, o printf continuaria imprimindo letras, mesmo que o vetor já tivesse acabado (ele não tem como saber o tamanho do vetor), até encontrar um sinal \0 (em algum outro lugar de memória).

Devido a isso, sempre que você for criar um vetor para armazenar uma string, pense no espaço adicional necessário ao terminador. Por exemplo, se você pensa em armazenar nomes com 30 letras cada, não se esqueça que necessitará de um espaço adicional para o \0. Sendo assim, seu vetor deverá ter 31 espaços.

Alguns exemplos com vetores:

Supondo que a variável numeros tenha sido declarada como um vetor de 10 elementos e que esses elementos tenham sido inicializados da seguinte forma:

Elemento:	6	10	20	31	40	17	60	8	7	90
Índice:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Qual seria o resultado de:

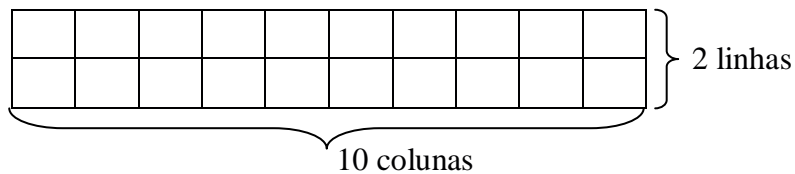
```
printf("%d", numeros[6]); // 60
scanf("%d", &numeros[5]); // colocaria o valor lido na posição 5
scanf("%d", &numeros[10]); // erro, não há posição 10!
```

10 TRABALHANDO COM MATRIZES

Em C as matrizes são similares aos vetores, basta definir a outra dimensão em um colchete adicional:

```
int m[2][10];
```

No exemplo anterior, a variável m cria uma matriz de duas linhas com dez colunas cada.



Logo, para criar uma matriz, basta especificar no primeiro colchete o número de linhas e no segundo o número de colunas. Para acessar um elemento da matriz, basta especificar a linha e a coluna (os índices) do elemento desejado. Os índices de cada elemento da matriz anterior são os seguintes:

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]	[0][8]	[0][9]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]	[1][8]	[1][9]

Logo, o código:

```
m[0][8] = 15; // coloca o valor 15 no na coluna 8 da linha 0
m[1][9] = 9;  // coloca o valor 9 no na coluna 9 da linha 1
m[0][0] = 10; // coloca o valor 10 no na coluna 0 da linha 0
m[1][4] = 4;  // coloca o valor 4 no na coluna 4 da linha 1
```

Deixando a matriz assim:

10	*	*	*	*	*	*	*	15	*
*	*	*	*	4	*	*	*	*	9

Agora você pode criar vetores de strings (que na verdade são matrizes de chars):

```
char listaNomes[10][31]; // cria lista de 10 nomes c/30 caracteres cada um
                          // (o trigésimo primeiro é para o \0 !!!)
```

L	E	O	N	A	R	D	O	\0	...	*
V	A	G	N	E	R	\0	*	*	...	*
F	A	B	I	O	\0	*	*	*	...	*
:										
G	I	S	L	A	I	N	E	\0	...	*

Se, por acaso, você desejar criar uma matriz de *strings*, deverá adicionar mais uma dimensão:

```
char listaNomes[10][10][31]; // 10 linhas com 10 nomes de 30 letras
```

11 TRABALHANDO COM PONTEIROS

Os ponteiros são variáveis que armazenam endereços (posições de memória) ao invés de valores. A declaração de um ponteiro é muito semelhante à declaração de uma variável comum. Numa variável comum, coloca-se o tipo de dado que a variável vai manipular, seguido de seu nome. Em um ponteiro, basta colocar o sinal * após o tipo da variável.

Exemplo:

```
int variavel; // cria uma variável normal para um inteiro
int* ponteiro; // cria um ponteiro para um inteiro
```

Veja a diferença entre a utilização de variáveis normais para ponteiros (as variáveis armazenam valores e os ponteiros armazenam endereços de variáveis):

```
char b;
int a;
char* pont_char;
int* pont_int;

a = 10; // Coloca o valor 10 em a
b = 'A'; // Coloca o valor 'A' em b

pont_char = &b; // Coloca o endereço de b em pont_char
pont_int = &a; // Coloca o endereço de a em pont_int
```

endereço	memória	
1000	'A'	b
1001		
1002	10	a
1003	1000	pont_char
1004	1001	pont_int
1005	:	

Lembre-se de que o símbolo & é um operador que retorna o endereço de uma variável. Por isso, pont_char recebe o endereço de b.

Note ainda que existem ponteiros específicos para cada tipo de dado. O ponteiro para *char* (pont_char) aponta para variáveis do tipo *char*, enquanto que o ponteiro para *int* (pont_int) aponta para variáveis do tipo *int*. Na verdade, os ponteiros podem apontar para qualquer tipo de variável. O C não faz restrições quanto a isso. Porém, dependendo do que você vai fazer com o ponteiro, pode ser que seu programa não funcione. Essa questão vai ser abordada com mais detalhes na seção seguinte – Aritmética de Ponteiros.

Se você mandar imprimir o conteúdo de um ponteiro, o que vai ser mostrado é o endereço que ele aponta:

```
printf("%d", pont_char); // imprime 1000
```

Você pode mandar o C imprimir o conteúdo do endereço que o ponteiro aponta, ou seja, o conteúdo da variável que o ponteiro contém o endereço. Para fazer isso, você deve utilizar o operador de *de-referência/indireção* (que é o *) antes da variável:

```
printf("%c", *pont_char); // imprime o dado que está no endereço
// apontado pelo ponteiro
```

11.1 Aritmética de ponteiros

Apesar de armazenarem endereços de memória os ponteiros também podem ser utilizados em expressões aritméticas. Você pode, portanto, somar, subtrair, multiplicar e dividir o endereço armazenado por um ponteiro. Só não esqueça que você estará fazendo isso com endereços de memória.

Essa característica é extremamente útil. Você poderia fazer o ponteiro apontar para a posição inicial de um vetor. Ao somar um ao endereço do ponteiro, automaticamente ele passa a apontar para o próximo elemento do vetor. Assim, é possível modificar o conteúdo de um vetor através de um ponteiro:

```
int a[5];
int* p;

p = &a; // inicializa o ponteiro no 1º elemento do vetor a

for(int contador = 0; contador < 5; contador++) { //do primeiro ao último
    *p = 0; // inicializa o elemento atual em zero
    p++; // passa para o próximo elemento
}
```

Ao lado você pode enxergar o vetor *a* na memória de um computador. Note que cada componente do vetor é um número inteiro que ocupa duas posições de memória (dois bytes). Se for assim, como é que a instrução *p++* (que é a mesma coisa que *p = p + 1*) consegue pular dois endereços de memória (e passar para o próximo elemento)?

Isso funciona porque o programador disse para o C que o ponteiro é um ponteiro do tipo `int` (`int* p`). O C também sabe que o tipo `int` ocupa dois espaços de memória. Portanto, quando uma operação matemática é feita em um ponteiro, o C leva em conta o tamanho do tipo para o qual o ponteiro foi declarado e passa corretamente para o próximo elemento.

endereço	memória	
1000	0	a[0]
1001		
1002	0	a[1]
1003		
1004	0	a[2]
1005		
1006	0	a[3]
1007		
1008	0	a[4]
1009		

No exemplo anterior, quando o endereço é incrementado em um, ele é incrementado em *um inteiro*, e passa para o endereço do próximo número inteiro que está dois bytes adiante. Nesse exemplo, os valores de *p* em cada iteração (passada) do laço são: 1000, 1002, 1004, 1006 e 1008.

Logo, na aritmética de ponteiros o endereço é incrementado (ou decrementado) de acordo com o tamanho do tipo de dado para o qual ele foi definido.

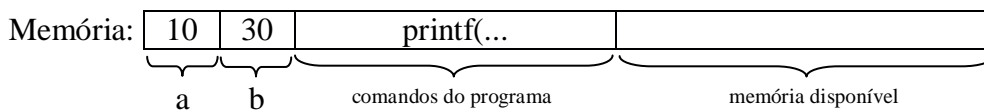
No exemplo anterior, se o ponteiro tivesse sido declarado como um ponteiro de *char* (`char* p`), os endereços de *p* teriam sido 1000, 1001, 1002, 1003 e 1004 (note que um *int* tem espaço para armazenar dois *chars*). O C deixaria isso funcionar, mas provavelmente os resultados poderiam ter sido inesperados para o programador.

12 ALOCAÇÃO DINÂMICA DE MEMÓRIA

Quando um programa é executado, todas as variáveis que foram declaradas pelo programador são criadas. O espaço de memória correspondente ao que cada uma delas precisa para funcionar é alocado. Esse espaço após ter sido alocado não muda mais. Diz-se que esse tipo de alocação é estático (já que não muda):

```
#include<stdio.h>
void main (void)
{
    int a = 10;
    int b = 30;

    printf("%d", a+b);
}
```



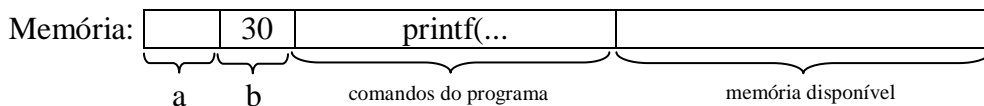
Porém, em alguns casos, o programador pode necessitar de mais espaço de memória durante a execução do programa. Na verdade isso é possível, porque os computadores possuem uma capacidade grande de memória, capaz de armazenar o programa, suas variáveis e ainda sobrar espaço.

Quando o programador muda o tamanho de uma variável ou aloca um espaço de variável que não é fixo (e pode variar esse espaço a cada execução do programa), diz-se que ele está fazendo alocação dinâmica de memória.

Em C, a alocação dinâmica de memória é feita com o auxílio de ponteiros. Os ponteiros apontam para qualquer endereço de memória. O espaço de memória onde o ponteiro (o endereço) vai ficar é alocado estaticamente, como qualquer outra variável, durante a inicialização do programa (o que vai variar é o local para o qual ele aponta):

```
#include<stdio.h>
void main (void)
{
    int *a;
    int b = 30;

    printf("%d", b);
}
```



Depois do programa ter sido carregado em memória, o ponteiro pode apontar para a memória restante e utiliza-la para armazenar outros valores e variáveis. Para poder utilizar uma região de memória livre (disponível) é preciso chamar uma função do sistema operacional que identifica um espaço de memória disponível aloca esse espaço para uma variável. O C possui uma função específica para isso, não sendo necessário utilizar uma função do sistema operacional. Essa função é a *malloc*.

A função `malloc` possui a seguinte sintaxe:

```
<endereço_livre> malloc(<espaço_desejado>);
```

Ou seja, basta passar como parâmetro o tamanho de espaço de memória que se deseja alocar que a função `malloc` encontra esse espaço (se ele existir) e retorna o endereço onde ela o encontrou.

Portanto, para alocar dinamicamente uma região de memória para uma variável do tipo `char`, basta fazer o seguinte:

```
#include<stdio.h>
#include <alloc.h> // biblioteca do TurboC que contém a função malloc

void main(void)
{
    char *p;
    p = (char*) malloc( 1 ); // aloca espaço para um char
}
```

Note o `(char*)` antes da função `malloc`. Ele é utilizado porque a função `malloc` retorna um endereço de memória qualquer, que pode ser utilizado por qualquer tipo de variável. Porém, nosso ponteiro é do tipo `char`. Portanto, o programador deve informar para o C que a função `malloc` está retornando o endereço para um tipo específico, no caso um `char`. Isso é feito pelo `(char*)`, que é um modificador de tipo (*type cast*).

No programa anterior:

```
#include <alloc.h> // biblioteca do TurboC que contém a função malloc
#include<stdio.h>

void main (void)
{
    int *a;
    a = (int*) malloc( 2 ); // o int ocupa 2 bytes
    *a = 10;
    printf("%d", *a);
}
```



Nesse exemplo, a função `malloc` retornou o endereço do primeiro local livre na memória disponível. Esse endereço foi passado para o ponteiro `a`. Preste atenção ao modificador de tipo `(int*)`. Ele sempre é necessário. Note que o tamanho de um `int` é *2 bytes*, logo, não esqueça de passar corretamente o espaço de memória que você vai precisar para armazenar seu dado!

Quando o programador atribuiu 10 ao local apontado por `a`, o número 10 foi colocado no endereço que a variável `a` apontava (o endereço que `malloc` retornou: o primeiro local livre).

A função *malloc* aloca qualquer espaço de memória (desde que exista esse espaço). Portanto, você pode utilizar *malloc* para alocar memória para um vetor:

```
#include <alloc.h>
#include<stdio.h>

01 void main (void)
02 {
03     int *vetor; // cria um ponteiro para um vetor
04     int total; // cria uma variável para armazenar o tamanho do vetor
05
06     printf("Quantos elementos? "); // pergunta para o usuário
07     scanf("%d", &total); // quantos elementos ele quer
08
09     vetor = (int *) malloc (total * sizeof(int));
10
11     for(int contador = 0; contador < total; contador++)
12     {
13         printf("Digite o %do elemento do vetor: ", contador);
14         scanf("%d", &vetor[contador]);
15     }
16
17     free(vetor);
18 }
```

O exemplo anterior pergunta para o usuário quantos elementos o vetor vai precisar. Com isso, esse valor pode ser utilizado para alocar dinamicamente um espaço de memória que consiga armazenar todo o vetor.

Pergunta: no exemplo anterior, na linha 9, por que foi passada a expressão `total*sizeof(int)` ao invés de `total`?

*Total é a variável que contem o número de elementos do vetor. Porém, como se deseja criar um vetor de inteiros (int), devemos multiplicar pelo tamanho do tipo int, que é 2. Logo, se passássemos total * 2 para a função malloc, ela já funcionaria corretamente. Porém, o C possui uma função que retorna o tamanho (em bytes) de um tipo de variável – a função sizeof (sizeof, em inglês, significa tamanho de). Assim, não precisamos decorar o tamanho de cada tipo de variável, basta escrever sizeof(int) que ela retorna o tamanho correto (que é 2). Além disso, o tamanho que cada tipo de variável ocupa pode variar de um tipo de processador para outro. Logo, a utilização de sizeof garante que estamos alocando o espaço correto para os elementos do vetor.*

Vimos então que essas duas funções (malloc e sizeof) funcionam em conjunto, e ajudam muito o programador.

Pergunta: Sabendo disso, como fazer para alocar dinamicamente um espaço de memória para armazenar um vetor de 30 números reais?

Basta fazer: `float *p = (float*) malloc(30 * sizeof(float));`

Ou seja, *p* recebe o endereço de um vetor de *floats* cujo tamanho é trinta vezes o tamanho de (float).

A função **free** que aparece na linha 17 do exemplo anterior merece uma atenção especial. Quando a variável ou vetor (cujo espaço foi alocado com a função malloc) não é mais necessário, o programador deve liberar seu espaço de memória. Essa função serve para isso, ela libera o espaço de memória que foi alocado para uma variável ou vetor através da função *malloc*. Essas duas funções sempre andam juntas!

Se você estiver utilizando um compilador compatível com o C++ (como o Turbo C, CBuilder ou o Visual C++) não é preciso utilizar a função *malloc*. Você pode utilizar uma forma mais natural, um comando introduzido pelo C++: o comando **new**.

Substitua a linha 9 do programa anterior por a linha seguinte e veja o resultado:

```
vetor = new int[total]; // cria um novo vetor de inteiros
```

Sempre que o comando *new* é utilizado (e não necessitamos mais utilizar a variável ou vetor) o operador **delete** deve ser utilizado:

```
delete[] vetor; // libera o espaço alocado para o vetor
```

13 SOBRE FUNÇÕES...

Imagine que você precise fazer um programa que colete os dados do usuário e coloque em uma variável. Tradicionalmente isso é feito através da função `scanf`. Essa função, porém, possui algumas limitações. Um exemplo de limitação dessa função consiste em ela não oferecer ao programador uma maneira dele especificar o tamanho máximo do que o usuário pode escrever na hora de digitar algum dado.

Suponha que você deseje que o usuário digite um nome de até 10 caracteres. O `scanf` não tem como limitar o tamanho da entrada de dados. Se o usuário digitar 15 ou mais caracteres, o `scanf` vai tentar colocar todos eles na variável que você passou-lhe como parâmetro. Se essa variável tiver sido definida como capaz de armazenar somente 10 caracteres, os caracteres restantes (do 11º ao 15º) vão acabar ocupando uma região de memória que não faz parte da variável (e isso pode gerar algum problema).

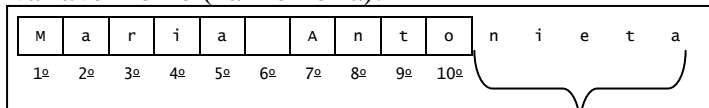
Programa:

```
#include <stdio.h>
void main(void)
{
    char nome[10];
    printf("Digite o no
gets(nome);
    :
```

Nome digitado pelo usuário (Tela do programa):

Digite o nome: Maria Antonieta

variável nome (na memória):



Espaço adicional
(região de memória que
não pertence à variável)

Para solucionar esse problema você poderia criar um trecho de código que vai coletando caracteres do teclado até o máximo desejado. Após, esses caracteres são colocados na variável corretamente:

```
#include <stdio.h>
void main(void)
{
    char nome[10]; int tamanho = 0;
    printf("Digite o nome: ");
    do{
        nome[tamanho] = getch(); // lê um caractere e coloca no vetor
        tamanho++; // incremente a posição atual do vetor
    }while(tamanho < 10); // enquanto não chegar ao final do vetor
    // sai do laço quando o usuário digitar a 10ª letra do nome
}
```

Agora suponha que você tenha que coletar diversos dados do usuário em locais e variáveis diferentes. Você teria que repetir o mesmo código diversas vezes:

```
#include <stdio.h>
void main(void)
{
    char nome[10], endereço[20], bairro[10];
    int pos;

    printf("Digite o nome: "); pos = 0;
do{
    // lê o nome:
    nome[pos] = getch(); // lê um caractere e coloca no vetor
    pos++;
    // incremente a posição atual do vetor
}while(pos < 10); // enquanto não chegar ao final do vetor
// sai do laço quando o usuário digitar a 10ª letra do nome

    printf("Digite o endereço: "); pos = 0;
do{
    // lê o endereço:
    endereco[pos] = getch(); // lê um caractere e coloca no vetor
    pos++;
    // incremente a posição atual do vetor
}while(pos < 20); // enquanto não chegar ao final do vetor
// sai do laço quando o usuário digitar a 20ª letra do endereço

    printf("Digite o bairro: "); pos = 0;
do{
    // lê o bairro:
    bairro[pos] = getch(); // lê um caractere e coloca no vetor
    pos++;
    // incremente a posição atual do vetor
}while(pos < 10); // enquanto não chegar ao final do vetor
// sai do laço quando o usuário digitar a 10ª letra do bairro
}
```

Nesse exemplo, o trecho marcado é repedido diversas vezes, com algumas pequenas alterações. Note como o programa fica grande e complexo, difícil de ser trabalhado, compreendido e corrigido (caso possua algum erro).

As funções resolvem esse problema. Uma função é um trecho de código que pode ser utilizado diversas vezes em um programa (ou diversos programas). Você define o trecho de código que deve ser repedido dentro de uma função. Depois, para que ele seja executado, basta chamar a função que ele é executado. Assim você não necessita escrever o mesmo código diversas vezes.

Para que a função (trecho de código) possa ser utilizada em variáveis diferentes ela deve ser flexível. Para conseguir essa flexibilidade, você pode parametrizá-la, isto é, chamar o código da função com valores ou atributos diferentes. Assim, ela pode funcionar de modo diferente ou para variáveis diferentes sem ter que ser alterada.

```

#include <stdio.h>
// Declara uma função que recebe como parâmetro um vetor do tipo char
// e seu tamanho (número de caracteres a serem lidos)
// A função preenche o vetor com os dados digitados pelo usuário:
void le_variavel(char variavel[], int tamanho)
{
    int pos = 0;
    do{
        variavel[pos] = getch(); // lê um caractere e coloca no vetor
        pos++;                  // incremente a posição atual do vetor
    }while(pos < tamanho);      // enquanto não chegar ao final do vetor
}

void main(void)
{
    char nome[10], endereço[20], bairro[10];

    printf("Digite o nome: ");   le_variavel(nome, 10);
    printf("Digite o endereço: "); le_variavel(endereco, 20);
    printf("Digite o bairro: "); le_variavel(bairro, 10);
}

```

Note no exemplo que o código que se repetia foi colocado dentro de uma função. Quando o código é necessário, o nome da função (`le_variavel`) é colocado no programa. O programa principal fica bem menor e mais fácil de ser compreendido. Se a função possuir um erro ou tiver que ser modificada, isso é feito em um único local (dentro da declaração da função). Todos os locais que chamam a função acabam sendo afetados pelas correções.

13.1 Definindo funções

Funções em C são como funções matemáticas:

$$x = f(y)$$

As funções (f) podem receber parâmetros (y) necessários para que realize a computação ou tarefa nela especificada (dependendo da tarefa pode não ser necessário um parâmetro). As funções também podem retornar um valor (x), que é o resultado de sua computação e que pode ser utilizado por uma outra função ou comando da linguagem.

Portanto, levando essas considerações em conta, a sintaxe para declarar uma função em C é a seguinte:

```
<tipo_de_retorno> nome_da_função(<parâmetro_1>, <parâmetro_2>, ... <parâmetro_n>);
```

Esse é chamado cabeçalho ou protótipo da função, que especifica seu nome, parâmetros e tipo de retorno. O nome da função segue as mesmas regras de nomes de variáveis: não pode ser uma palavra reservada e pode conter letras e números (a partir da segunda letra). O tipo de retorno é um dos tipos de variáveis do C (int, float, long, double...) ou algum tipo definido pelo usuário (uma struct). Os parâmetros são as variáveis que a função recebe para poder funcionar. Os parâmetros devem ter especificados os seus tipos.

Exemplos:

```
int soma(int a, int b); → Declara uma função que retorna um número inteiro e recebe dois parâmetros: a e b que são variáveis que armazenam números inteiros.

void le(char v[], int t); → Declara uma função que não retorna valor (void) e recebe dois parâmetros: um vetor de tamanho variável v (que vai ter o tamanho da variável passada pelo programador na chamada da função) e um número inteiro t (indicando o tamanho do vetor).

void menu(void); → Declara uma função que não retorna valor (void) e que não recebe parâmetros. Neste caso, ela realiza um procedimento que não necessita de parâmetros (desenha um menu na tela).

int getch(void); → Essa função não recebe parâmetros mas retorna um número inteiro que indica o código de tecla pressionada pelo usuário.
```

Agora que já aprendemos a declarar uma função (ou seja, já sabemos como definir os parâmetros de que ela precisa para funcionar e seu tipo de retorno), resta aprender como implementar o corpo da função, isto é, colocar o trecho de código que ela deve executar quando chamada. Vamos dar uma olhada na função *le_variavel* utilizada como exemplo anteriormente:

Cabeçalho (declaração)	<pre>void le_variavel(char variavel[], int tamanho) {</pre>
Corpo (trecho de código executado quando ela é chamada)	<pre> int pos = 0; do{ variavel[pos] = getch(); // lê um caractere e coloca no vetor pos++; // incremente a posição atual do vetor }while(pos < tamanho); // enquanto não chegar ao final do vetor }</pre>

O cabeçalho desta função indica que ela realiza o processamento interno (descrito no corpo dela) e não retorna nenhum valor para quem a chamou (void). Após, o mesmo cabeçalho descreve duas variáveis: variavel[] (que é um vetor de chars) e tamanho (que é um int). Essas duas variáveis são utilizadas no processamento descrito no corpo da função.

Quando uma função necessita retornar um valor para quem a chamou, torna-se necessário definir no seu cabeçalho o tipo de valor que vai ser retornado (int, float, double...). O programa seguinte contém uma função que soma dois valores (que são passados como parâmetros para ela) e retorna o resultado dessa soma para quem a chamou:

```

#include <stdio.h> // biblioteca contendo a descrição das funções de E/S
// (tais como printf e scanf)

int soma(int a, int b) // declara uma função que recebe dois parâmetros
// inteiros e retorna um valor que é a soma deles
{
    int resultado;
    resultado = a + b;
    return resultado; // retorna o resultado da soma
}

void main(void)
{
    int a1, b1, r;
    a1 = 10; b1 = 20;
    r = soma(a1, b1); // coloca o valor retornado pela função em r
    printf("%d\n", r); // mostra o conteúdo de r
    printf("%d\n", soma(30, 15)); // imprime o valor retornado pela função
}

```

Note que a função soma justamente soma o conteúdo das variáveis a e b (que contêm as variáveis ou valores passados como parâmetros na chamada da função no programa principal) e coloca esse resultado na variável **resultado**. Depois o conteúdo dela é retornado. Quem informa qual valor deve ser retornado é o comando **return**. Sempre que você desejar retornar um valor para quem chamou a função, coloque esse comando. O valor retornado tem que ser do mesmo tipo para o qual a função foi declarada (no caso um **int**). Se a função não retornar nada (isto é, tiver sido declarada como **void**) o return não aparece ou, se aparecer, deve aparecer sozinho:

```

void le_variavel(char variavel[], int tamanho)
{
    int pos = 0;
    do{
        variavel[pos] = getch(); // lê um caractere e coloca no vetor
        pos++; // incremente a posição atual do vetor
    }while(pos < tamanho); // enquanto não chegar ao final do vetor
    return; // indica que a função terminou e não retornou nada!
}

```

13.2 Escopo de variáveis: globais e locais

O escopo de uma variável nada mais é do que o local onde ela é válida ou reconhecida. Quando uma variável é declarada, ela existe e tem validade dentro do bloco ou função para o qual ela foi declarada. Fora desse bloco ou função ela não pode ser utilizada e não é reconhecida:

```

void main(void)
{
    int var_a; // declara uma variável local
}

void funcao_x(void)
{
    int var_b; // declara uma variável local
}

```

A **var_a** só tem validade (só funciona) dentro da função **main**, e a **var_b** só tem validade dentro da **funcao_x**. Isso significa que o programador não pode utilizar a variável **var_a** dentro da **funcao_x**:

```
void main(void)
{
    int var_a;
    var_b = 5; // Errado! A variável só existe dentro de funcao_x
}

void funcao_x(void)
{
    int var_b;
    var_a = 10; // Errado! A variável só existe dentro de main
    var_b = 10; // Correto, a variável pertence à funcao_x
}
```

Devido a isso, o programador pode, inclusive, declarar duas ou mais variáveis de mesmo nome, desde que cada uma pertença a uma função diferente. É como se cada função tivesse a sua variável desconhecida pelas outras funções:

```
void main(void)
{
    int var_a = 5;
    funcao_x();
    printf("var_a do main vale: %d", var_a); // vai mostrar 5
}

void funcao_x(void)
{
    int var_a = 10;
    printf("var_a de funcao_x vale: %d", var_a); // vai mostrar 10
}
```

No exemplo anterior, as duas variáveis (**var_a**) são diferentes!

Você pode criar variáveis que sejam acessíveis (conhecidas) por todas as funções do programa. Essas variáveis são chamadas de variáveis globais. Para fazer isso, você deve declarar uma variável logo no início do programa, fora de qualquer função:

```
#include <stdio.h>
int c ; // declara uma variável global

void main(void)
{
    c = 10; // coloca o valor 10 na variável global
    printf("%d", c); // vai mostrar 10
    f1();
    printf("%d", c); // vai mostrar 15
    c = 20; // coloca o valor 20 na variável global
    f2();
}
```

```

void f1(void)
{
    c = 15; // coloca o valor 15 na variável global
}

void f2(void)
{
    printf("%d", c); // vai mostrar 20
}

```

13.3 Considerações sobre a passagem de parâmetros

Quando uma função é chamada os seguintes passos ocorrem:

- Criação das variáveis especificadas como parâmetro (essas variáveis são criadas como se fossem variáveis locais da função);
- Inicialização das variáveis através da atribuição dos valores passados como parâmetros para elas.

Vamos ver um exemplo:

```

void f1(int a, char b) // declaração da função
{
    // alguns comandos
}

```

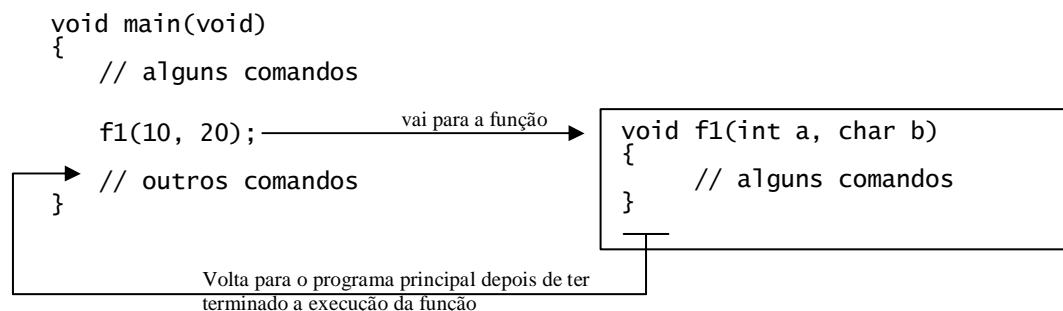
Quando essa função é chamada:

```

void main(void)
{
    // alguns comandos
    f1(10, 20); // chama a função f1
    // outros comandos
}

```

No momento em que a execução do programa chega na linha em que a função f1 é chamada, o programa vai para o cabeçalho da função e cria as duas variáveis (a e b). Depois de as ter criado, o valor 10 (que é o primeiro parâmetro) é atribuído a primeira variável (a) e o valor 20 é atribuído a segunda variável (b):



Assim, enquanto o programa estiver dentro da função, as variáveis a e b vão existir e seus valores vão ser inicializados nos valores que foram passados por parâmetro na chamada da função.

Pode-se dizer que:

```

void f1(int a, char b)

```

É o mesmo que:

```
void f1()
{
    int a;
    char b;
    // demais comandos da função
}
```

Porque as duas variáveis são criadas normalmente, como se tivessem sido declaradas como variáveis locais da função. A diferença é que no primeiro caso as variáveis são inicializadas com os valores passados pelo programador na chamada da função, e no segundo não. Pegando o exemplo anterior, onde o programador chamava a função `f1` com os parâmetros 10 e 20:

```
f1(10, 20)
```

Seria o mesmo que fazer explicitamente:

```
void f1()
{
    int a = 10;
    char b = 20;
    // demais comandos da função
}
```

Você também pode passar variáveis para funções (ao invés de valores):

```
void main(void)
{
    int v1 = 30, v2 = 40;
    f1(v1, v2);
}
```

Neste caso, somente o conteúdo das variáveis é passado como parâmetro, não as variáveis em si. Isso significa que as variáveis `a` e `b` da função `f1` vão ser criadas normalmente e inicializadas com os conteúdos das variáveis passadas como parâmetro:

```
void f1(int a, char b);
```

A variável **a** vai ser uma variável local (completamente independente da variável `v1`) cujo valor vai ser inicializado ao valor que `v1` tinha no momento da chamada da função (o valor 30). Da mesma forma, a variável **b** vai ser inicializada em 40.

Se internamente a função mudar o valor de suas variáveis os valores das variáveis externas vão continuar sendo os mesmos. Justamente por serem variáveis locais, as variáveis da função não tem nada a ver com as externas. As externas só servem (nesse caso) para passar o valor inicial das variáveis locais (internas da função):

```
void main(void)
{
    int v1 = 30, v2 = 40;
    f1(v1, v2);
    printf("%d - %d", v1, v2); // mostra 30 - 40
}
void f1(int a, char b)
{
    a = b + 10; // não altera v1 (pois são variáveis independentes)
    b = 15;    // não altera v2 (pois são variáveis independentes)
}
```

Esse tipo de passagem de parâmetros é chamado de passagem por valor, pois somente o valor de uma variável é passado para a variável local (interna) da função. A passagem de parâmetros por valor não gera nenhuma dependência entre a variável externa (passada como parâmetro) e a variável local (do cabeçalho) da função.

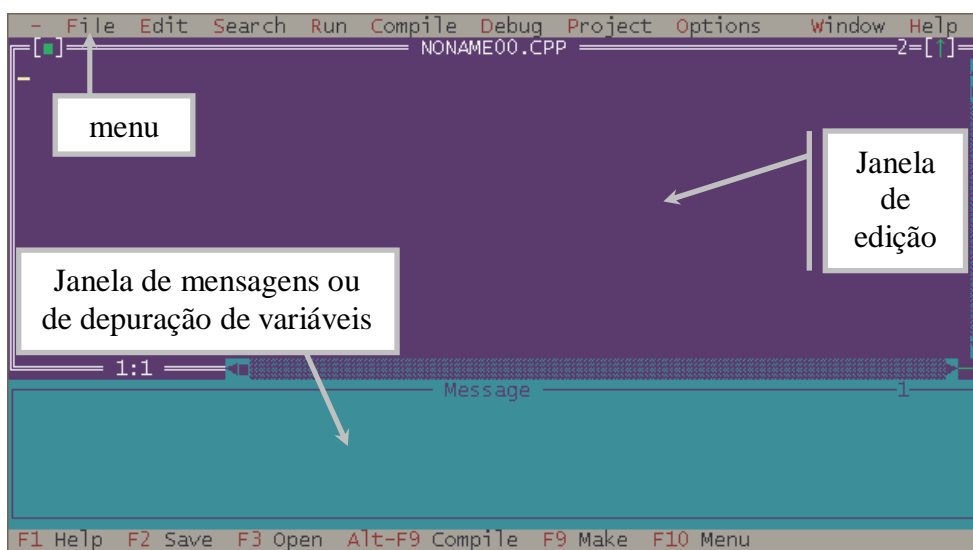
14 APÊNDICE I: MANUAL DO TURBO-C LITE (TCLITE)

O Turbo-C lite (pronuncia-se *tc laite*) é um ambiente de programação desenvolvido pela empresa Borland (<http://www.borland.com>). Por ser antigo, porém muito bom, esse ambiente foi liberado pela empresa para uso pessoal e educacional (não comercial). Ele é, na verdade, uma versão DEMO do ambiente Turbo-C original, que ainda é comercializado pela empresa. Por ser uma versão DEMO, o *tclite* não gera arquivos executáveis que possam ser executados fora do ambiente e/ou vendidos.

Os interessados em utilizar esse ambiente podem pegá-lo no museu da Borland na WEB. Para tanto basta realizar um cadastrando na *Borland Community* (comunidade Borland) que fica situada no endereço <http://community.borland.com>, entrar na seção *Museum* (museu) e selecionar o arquivo “tclite.exe” para *download*. Neste museu a Borland também disponibiliza uma série de outros ambientes não mais comercializados por ela.

Apesar de antigo o Turbo-C é um dos melhores ambientes de programação para MS-DOS já desenvolvidos. Ele integra um editor de programas, um compilador e um depurador (*debugger*) que permite a execução passo-a-passo dos programas criados pelo programador, além de possibilitar a visualização do conteúdo de cada variável a qualquer momento de tempo.

14.1 O ambiente tclite



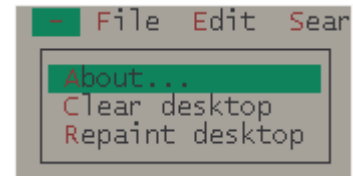
A figura acima mostra a janela básica do ambiente *tclite*. Podemos dividi-la em três regiões principais: o *menu*, a *janela de edição* e a *janela de mensagens ou de depuração de variáveis*. Vamos agora estudar em detalhes cada uma destas regiões.

14.2 O menu

É através do menu que você cria novos programas, salva-os, fecha-os e abre-os posteriormente. O menu também serve para uma série de outras coisas, como você vai ver a seguir. Para acessar o menu, pressione a tecla ALT esquerda mais a letra inicial do menu que você quer acessar (note, na figura, que todos os menus possuem sua letra inicial avermelhada, indicando que essa é a sua letra de acesso). Você também pode utilizar o mouse para acessar ao menu (se este estiver configurado corretamente).

14.2.1 O menu do sistema (-)

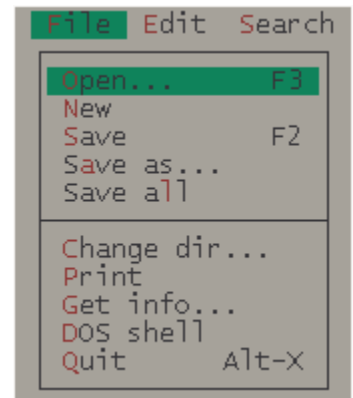
O menu do sistema fica situado do lado esquerdo superior da janela do ambiente, e é representado por um sinal de '-'. Neste menu você encontra as seguintes opções:



- **About... (sobre):** Mostra informações sobre o sistema (versão, fabricante...);
- **Clear desktop (limpar área de trabalho):** Fecha/esconde todas as janelas abertas;
- **Repaint desktop (repintar área de trabalho):** redesenha todas as janelas abertas.

14.2.2 O menu de arquivos (File) [ALT + F]

O menu de arquivos permite que você abra, crie, salve e imprima arquivos, entre outras coisas. Utilize o mouse ou as setas direcionais (←↑↓→) para escolher a opção que você deseja selecionar. A seleção atual permanece grifada de verde. Você também pode selecionar uma opção pressionando a sua letra de acesso (em vermelho).

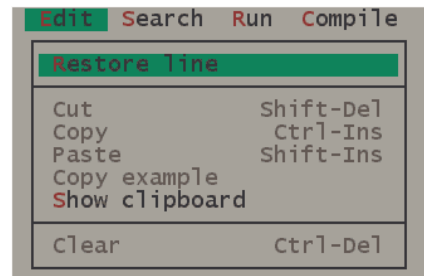


- **Open (abrir) [F3]:** Exibe uma caixa de diálogo solicitando o nome do arquivo que você deseja abrir. Nesta caixa há ainda uma região onde são listados todos os arquivos-fonte do diretório atual. Os arquivos-fonte geralmente possuem a extensão '.c' ou '.cpp'. Você seleciona com o mouse ou com as setas direcionais o arquivo a ser carregado. Se o arquivo não estiver no diretório ou *drive* atual, você pode mudá-los simplesmente digitando o nome do *drive* ou do diretório onde ele se encontra (a: ou c:\temp, por exemplo);
- **New (novo):** Abre uma nova janela no editor contendo um arquivo em branco. Esse novo arquivo recebe um nome provisório do tipo 'NONAME00.cpp' (que significa SEMNOME) até que seja salvo pela primeira vez. Ao salvar o arquivo pela primeira vez, você pode mudar seu nome para um mais adequado (se você não muda-lo, o *tclite* salva-o em 'noname00.cpp' e o próximo arquivo novo recebe o nome de 'noname01.cpp');
- **Save (salvar) [F2]:** Salva o arquivo atual. Se o arquivo estiver sendo salvo pela primeira vez, uma caixa de diálogo surgirá para você escolher o nome do arquivo (ou salvar com o nome sugerido 'nonamexx.cpp') e o seu local (drive e diretório). Se você utilizar o nome de um arquivo já existente o ambiente perguntará se você quer sobrescrever o antigo com o novo (o que o fará perder os dados do antigo). Neste caso, responda *No* (não) para a pergunta '*overwrite existing file?*' (sobrescrever arquivo existente?) se você não quiser perder o arquivo antigo ou *Yes* (sim) se você quiser gravar por cima dele;
- **Save as... (salvar como):** Permite que você salve o arquivo atual em um outro arquivo, que você deverá indicar o nome na caixa de diálogo que surgirá. Depois de salvo, você passará a trabalhar (modificar) o arquivo salvo (inclusive em um outro *drive* ou diretório), porém, o arquivo antigo continuará existindo;
- **Save all (salvar todos):** Salva todos os arquivos abertos;
- **Change dir... (trocar diretório):** Abre uma caixa de diálogo onde você pode mudar o drive e o diretório atual (onde os arquivos são salvos);
- **Print (imprimir):** Imprime o arquivo atual;
- **Get info... (pegar informações):** Abre uma janela com informações sobre o arquivo atual (número de linhas, diretório), memória utilizada e memória disponível, entre outras coisas.

- **DOS shell (prompt do MSDOS):** Sai momentaneamente para o MSDOS, permitindo com que você execute alguma tarefa ou comando do DOS. Para voltar ao *tclite*, digite *exit* no *prompt* do DOS e pressione [ENTER].
- **Quit (sair) [ALT + X]:** Sai do *tclite*. Se existir algum arquivo que ainda não tenha sido salvo, surgirá uma caixa de diálogo perguntando se você deseja salvá-lo. Responda *Yes* (sim) para salvar o arquivo, *No* (não) para perde-lo ou *Cancel* (cancelar) para continuar no ambiente.

14.2.3 Menu de edição (Edit)

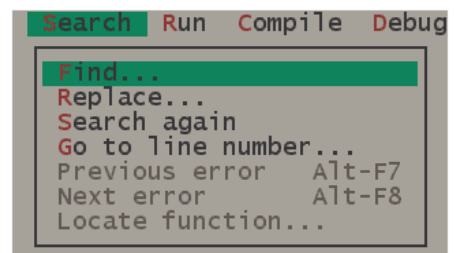
Oferece comandos de manipulação de texto. Muitos dos comandos só são ativados se você tiver marcado um bloco de texto. Para marcar um bloco, mantenha a tecla SHIFT pressionada e utilize as teclas direcionais para demarcar o bloco. Você também pode utilizar o mouse, mantendo pressionado o botão do mouse no local onde você quer marcar o início do bloco e arrastando-o até o local onde você quer marcar o fim do bloco. A região selecionada (bloco) fica marcada com um fundo cinza.



- **Restore line (restaurar linha):** Desfaz a última digitação;
- **Cut (recortar) [SHIFT+DEL]:** Como no Windows, recorta/apaga o bloco de texto selecionado.
- **Copy (copiar) [CTRL+INS]:** Copia o bloco selecionado para a área de transferência.
- **Paste (colar) [SHIFT+INS]:** Cola o bloco da área de transferência na posição atual do cursor.
- **Copy example (copiar exemplo):** Se você estiver no *Help* (ajuda), essa opção copia o exemplo de código apresentado para a área de transferência.
- **Show clipboard (mostrar área de transferência):** Abre a janela da área de transferência, mostrando todos os blocos já copiados até o momento. Nesta janela, você pode editar o bloco atual que será utilizado em uma ação de colar (paste) posterior.
- **Clear (limpar) [CTRL+DEL]:** Apaga o bloco selecionado.

14.2.4 Janela de Busca (Search)

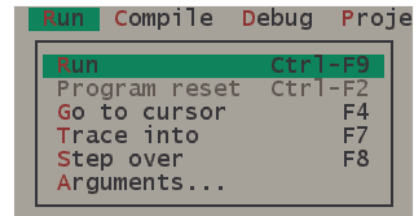
- **Find... (procurar):** Serve para você procurar algum texto (palavra, comando, nome de função) no seu arquivo. A busca é feita a partir do início do texto.
- **Replace... (substituir):** Serve para você substituir todas as ocorrências de uma palavra por outra no texto. Extremamente útil se você resolveu modificar o nome de uma variável, por exemplo.
- **Search again (procurar novamente/continuar a busca):** Uma vez que você tenha achado uma palavra ou texto com o comando *Find*, você pode utilizar o *Search again* para continuar procurando suas próximas ocorrências. O comando *Find* reiniciaria a busca, localizando sempre a mesma ocorrência da palavra. O comando *search again* continua a busca a partir do último local onde o texto foi encontrado, localizando suas próximas ocorrências.
- **Go to line number... (ir para a linha número):** Serve para você ir diretamente para determinada linha do programa.
- **Previous error (erro anterior) [ALT+F7]:** Se após ser compilado o programa possuir erros, esse comando passa diretamente para o próximo erro encontrado.



- **Next Error (próximo erro)** [ALT+F8]: Se após ser compilado o programa possuir erros, esse comando passa diretamente para o erro anterior.
- **Locate function... (localizar função)**: Após ter compilado um programa, esse comando permite você localizar/procurar uma função.

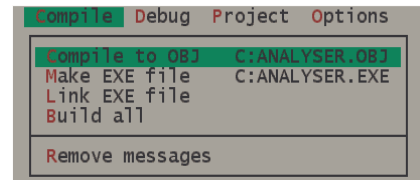
14.2.5 Menu de execução (Run)

- **Run (executar)** [CTRL+F9]: Executa/roda o programa atual. Compila-o se ele ainda não tiver sido compilado. Neste caso, se houverem erros o programa não é executado.
- **Program reset (finalizar programa)** [CTRL+F2]: Termina a execução/depuração de um programa.
- **Go to cursor (vai até o cursor)** [F4]: Executa um programa até a posição atual do cursor (e fica aguardando). Como em *Run*, compila o programa se ele ainda não tiver sido compilado.
- **Trace into (rastrear por dentro)** [F7]: Execução passo-a-passo. Executa 1 passo (uma linha de código) do programa, entrando dentro do código das funções. Ou seja, também executa passo-a-passo os comandos existentes dentro das funções que encontra pela frente.
- **Step over (rastrear por fora)** [F8]: Execução passo-a-passo. Executa 1 passo (uma linha de código) do programa, pulando as funções (executando-as por inteiro).
- **Arguments (argumentos)**: Permite que você defina os argumentos (de linha de comando) passados para o seu programa. Especialmente útil se você quiser fazer programas que recebam parâmetros e testá-los.



14.2.6 Menu de compilação (Compile)

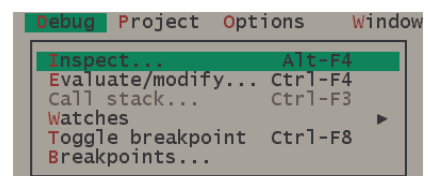
- **Compile to OBJ (compilar para OBJ)**: compila o programa e, se não houver erro, gera o arquivo OBJ (objeto) que corresponde ao programa executável sem as bibliotecas.
- **Make EXE file (gerar arquivo EXECUTÁVEL)**: gera o arquivo executável, compilando-o primeiro.
- **Link EXE file (ligar/linkar arquivo EXE)**: pega todos os arquivos OBJ relativos a um programa e junta-os em um arquivo executável.
- **Build all (construir tudo)**: re-compila todo o programa ou projeto atual e gera o arquivo executável se não for encontrado algum erro.
- **Remove messages (remover mensagens)**: limpa a janela de mensagens.



14.2.7 Menu de depuração (Debug)

Serve para você poder depurar o seu programa, executando-o passo-a-passo e verificando o conteúdo das variáveis a cada instante (para maiores informações, leia o anexo II: utilizando o depurador do Turbo-c)

- **Inspect... (inspecionar)** [ALT+F4]: abre uma janela onde você pode digitar o nome de uma variável. Após, todas as informações relativas a essa variável são mostradas (endereço, conteúdo...).
- **Evaluate/modify... (avaliar/modificar)** [CTRL+F4]: Você pode utilizar essa opção para fazer várias coisas: calcular expressões matemáticas (incluindo várias variáveis do

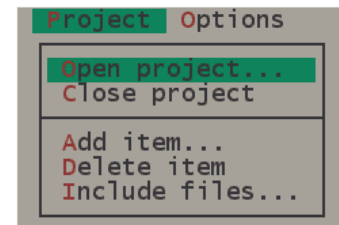


programa), verificar o conteúdo atual de uma variável e também, alterar o conteúdo de uma variável em tempo de execução.

- **Call stack... (pilha de chamada)** [CTRL+F3]: mostra a pilha de funções em execução no momento.
- **Watches (Variáveis):** abre um sub-menu onde você pode adicionar, remover e editar variáveis para monitoramento. O conteúdo destas variáveis é mostrado na janela de variáveis (watches), que só aparece se houverem variáveis sendo monitoradas (e é semelhante a janela de mensagens).
- **Toggle breakpoint (adicionar/remover ponto de parada)** [CTRL+F8]: Adiciona ou retira um ponto de parada na linha atual do cursor. Os pontos de parada são pintados de vermelho, e indicam ao compilador que o programa deve parar sua execução quando a linha do ponto de parada for atingida.
- **Breakpoints... (pontos de parada):** Abre uma janela contendo a lista de todos os pontos de paradas existentes no programa atual. Você pode incluir ou apagar pontos de parada, além de poder definir regras que informem ao compilador que o programa só deve parar em determinado ponto de parada depois de ter executado a linha onde ele se encontra n vezes, por exemplo.

14.2.8 Menu de projeto (Project)

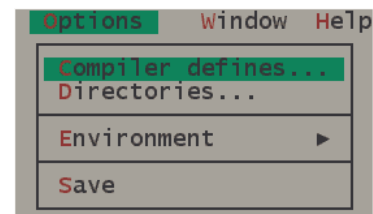
Essa opção serve para você criar programas que envolvam mais de um código-fonte ao mesmo tempo. Assim, após cadastrar todos os componentes envolvidos no seu programa (inclusive bibliotecas de outros fabricantes), o compilador pode gerenciar os diferentes códigos alterados e compila-los e juntá-los em um único arquivo executável.



- **Open project... (abrir/criar projeto):** Abre um projeto já existente ou cria um novo.
- **Close project (fechar projeto):** Fecha o projeto atual.
- **Add item... (adicionar item):** Adiciona um componente (arquivo de código-fonte ou biblioteca) ao projeto.
- **Delete item (apagar item):** Apaga (tira da lista) um componente de projeto.
- **Include files... (Arquivos incluídos no projeto):** mostra a janela contendo a lista de arquivos que fazem parte do projeto atual.

14.2.9 Menu de opções (Options)

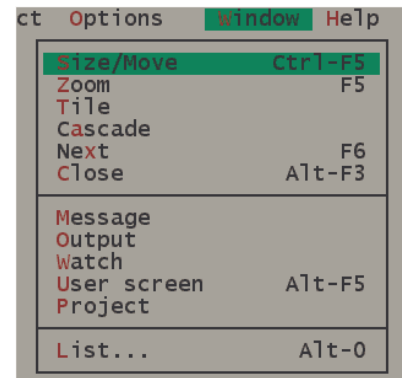
- **Compiler defines... (definições para o compilador):** Abre uma caixa de diálogo onde você pode incluir definições (idênticas aos #define colocados no código), indicando substituições ou macros para o pré-processador. Neste caso, você não precisa colocar o #define na frente das definições. Separe-as por “ ; ” (ponto e vírgula). Exemplo: DOIS = 2; TAMANHO = 10.
- **Directories... (diretórios):** Especifica os diretórios onde os arquivos do *tclite* necessários à compilação foram instalados. Há três opções:
 - **Include directories:** diretório onde se encontram os arquivos de cabeçalho de bibliotecas (arquivos “.h” ou “.hpp”), que você especifica com o comando `#include <arquivo.h>`. O diretório padrão é: `c:\tclite\include`.
 - **Library directories:** diretório onde se encontram as bibliotecas do *tclite* (biblioteca gráfica, de ponto flutuante, de memória, etc: arquivos “.lib”). O diretório padrão é: `c:\tclite\lib`.



- **Output directory:** diretório de saída. É o local onde os arquivos “.obj” e os arquivos executáveis vão ser colocados. Deixe em branco para que eles sejam colocados no mesmo diretório do programa-fonte.
- **Environment (ambiente):** Permite com que você especifique suas preferências do ambiente de programação, tais como: tamanho da tela (25 ou 50 linhas), arquivos que devem ser salvos quando você compila ou sai do ambiente (arquivos-fonte, configurações do ambiente, área de trabalho e projeto), opções do editor (criar arquivos de backup, modo de inserção ligado, utilizar tabulações...) e opções do mouse.
- **Save (salvar):** Salva as suas opções de ambiente.

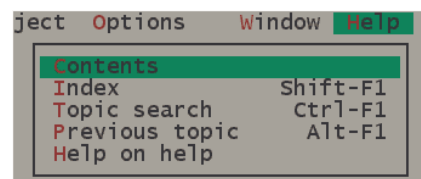
14.2.10 Menu de Janela (Window)

- **Size/move (tamanho/mover) [CTRL+F5]:** permite mover a janela atual para outra posição ou alterar o seu tamanho.
- **Zoom () [F5]:** faz a janela atual ficar do tamanho da tela.
- **Tile (lado-a-lado):** Se houverem várias janelas abertas ao mesmo tempo, coloca-as lado-a-lado (como no MS-Windows).
- **Cascade (cascata):** Se houverem várias janelas abertas ao mesmo tempo, alinha-as em forma de cascata (como no MS-Windows).
- **Next (próxima) [F6]:** Se houverem várias janelas abertas ao mesmo tempo, faz a próxima da lista tornar-se ativa (traz ela para a frente das outras).
- **Close (fechar) [CTRL+F3]:** Fecha a janela atual (ativa).
- **Message (mensagens):** Mostra a janela de mensagens.
- **Output (saída):** Mostra a janela de saída (mensagens geradas pelo seu programa, quando no modo texto, e a linha de comando/aviso do MS-DOS).
- **Watch (variáveis):** Mostra a janela de depuração de variáveis.
- **User screen (janela do usuário):** Mostra a janela onde o programa é executado (modo texto ou modo gráfico).
- **Project (projeto):** Mostra a janela de projeto.
- **List... (listar):** Abre uma lista com todas as janelas abertas no momento.



14.2.11 Menu de Ajuda (Help)

- **Contents (conteúdos):** Abre uma janela onde você pode navegar pelos conteúdos do arquivo de ajuda, divididos em diversas categorias.
- **Index (índice) [SHIFT+F1]:** Abre uma janela contendo o índice de todas as funções ou comandos da linguagem.
- **Topic search (busca por tópico) [CTRL+F1]:** Abre uma janela contendo a ajuda para o comando ou função onde o cursor está posicionado.
- **Previous topic (tópico anterior) [ALT+F1]:** Faz a janela de ajuda mostrar o tópico de ajuda visto anteriormente (extremamente útil quando se está navegando pelos tópicos de ajuda).
- **Help on help (ajuda da ajuda):** Ensina como utilizar o sistema de ajuda.



14.3 A janela de edição

É na janela de edição que você implementa (escreve) seus programas escritos na linguagem C. Ela é um editor de textos simples em recursos de formatação de texto (tais como negrito, itálico e seleção de tamanho e tipo de letras), mas estes recursos não são necessários para a edição de códigos de programas.

14.4 A janela de mensagens e depuração de variáveis

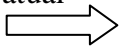
A janela de mensagens serve para o compilador informar para você os erros encontrados no seu programa e o status atual de alguma tarefa que ele esteja realizando (compilando, ligando...). Quando houver erros, simplesmente clique com o mouse em cima do erro que o editor abre a janela do código-fonte e posiciona-se em cima da linha onde ele ocorreu.

A janela de depuração de variáveis serve para você fazer a depuração do seu programa. Ao adicionar uma variável do seu programa nesta janela, você pode visualizar o seu conteúdo a cada passo de execução do programa. Para maiores informações sobre isso, consulte o Anexo seguinte, denominado Apêndice II: Utilizando o depurador do Turbo-c.

15 APÊNDICE II: USANDO O DEPURADOR DO TURBO-C

O ambiente de programação "Borland Turbo-C" permite que o programador execute o programa passo à passo. Para tanto, compile o seu programa normalmente (com a tecla F9) e, após, pressione a tecla F8. Geralmente, se você apertar a tecla F8 o Turbo-C deverá compilar o seu programa, não sendo necessário que você compile-o antes (tecla F9). A tecla F8 é a tecla de execução passo à passo. Para que você saiba em que ponto de execução seu programa está, o Turbo-C marca de azul a linha de execução atual (*step-point*).

Linha de
execução
atual



```
MS TC
Auto
File Edit Search Run Compile Debug Project Options Window Help
EXEMPLO6.CPP
printf("Contagem regressiva para o lançamento do foguete:\n");
printf("-----");

gotoxy(70, 23);
textcolor(GREEN); // muda a cor do texto para verde
cprintf("!*!");
gotoxy(1,24);
printf("

for(contador = 10; contador > 0; contador--)
{
    gotoxy(51, 2);
    printf("%5d", contador);
    delay(1000); // espera um segundo
}
25:1

Message
•Compiling C:\TEMP\ICLITE\BIN\EXEMPLO6.CPP:
Linking C:\TEMP\EXEMPLO6.EXE:
F1 Help F7 Trace F8 Step F9 Make F10 Menu
```

Cada vez que você pressiona essa tecla, o Turbo-C executa a próxima linha de comando do seu programa:

Quando F8 é
pressionada,
o Turbo-C
executa a linha
atual e passa
para a próxima
linha



```
MS TC
Auto
File Edit Search Run Compile Debug Project Options Window Help
EXEMPLO6.CPP
printf("Contagem regressiva para o lançamento do foguete:\n");
printf("-----");

gotoxy(70, 23);
textcolor(GREEN); // muda a cor do texto para verde
cprintf("!*!");
gotoxy(1,24);
printf("

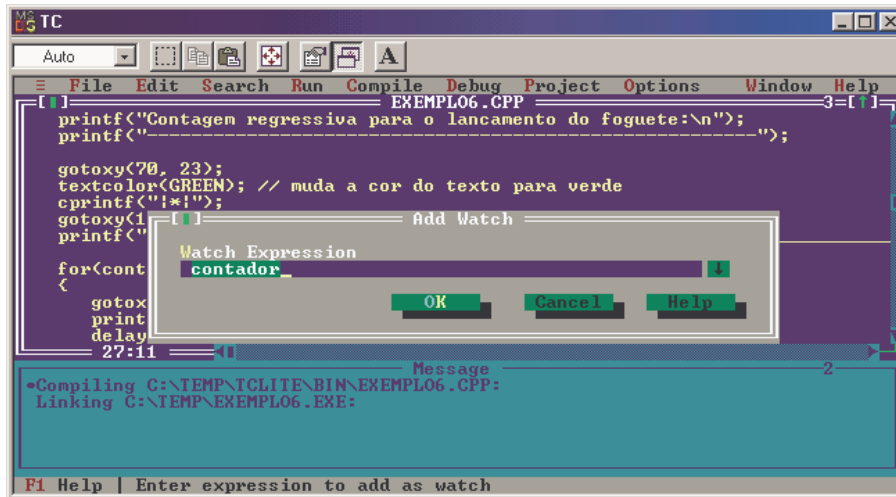
for(contador = 10; contador > 0; contador--)
{
    gotoxy(51, 2);
    printf("%5d", contador);
    delay(1000); // espera um segundo
}
27:1

Message
•Compiling C:\TEMP\ICLITE\BIN\EXEMPLO6.CPP:
Linking C:\TEMP\EXEMPLO6.EXE:
F1 Help F7 Trace F8 Step F9 Make F10 Menu
```

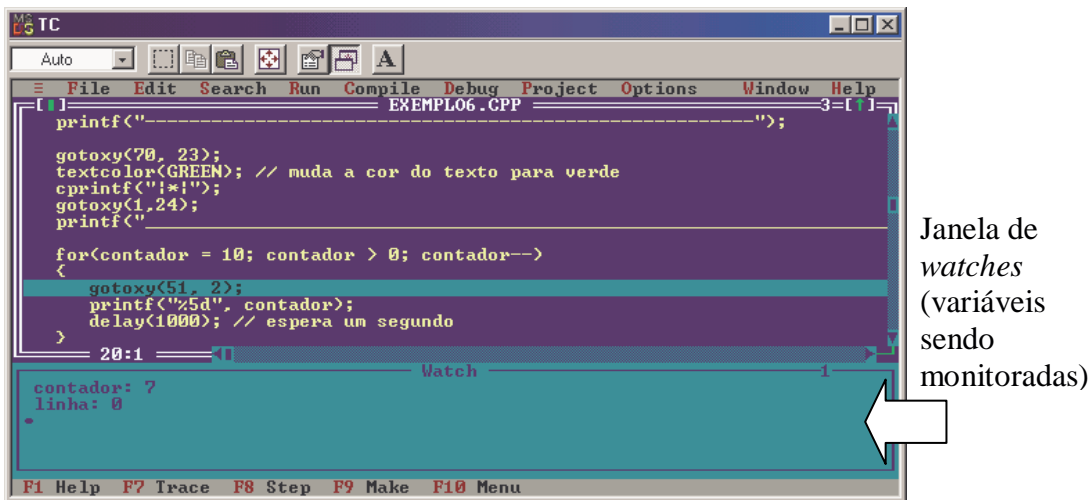
O programa vai sendo executado até que ele chegue ao seu final. Caso você queira interromper a execução atual do programa e reinicia-la por algum motivo, pressione CTRL-F2.

15.1 Verificando o conteúdo de variáveis

O Turbo-C permite que você veja o conteúdo das variáveis de programa que você declarou. Para isso, pressione CTRL-F7. A janela de adição de variáveis (*add watch*) aparecerá. Digite o nome da variável a ser monitorada:



A partir do momento em que a variável é declarada, o Turbo-C vai mostrando seu conteúdo na janela de *watches*. Assim, você pode identificar o conteúdo de uma variável e o seu comportamento ao longo da execução de um programa. Deste modo, você pode identificar eventuais erros de lógica no seu programa.

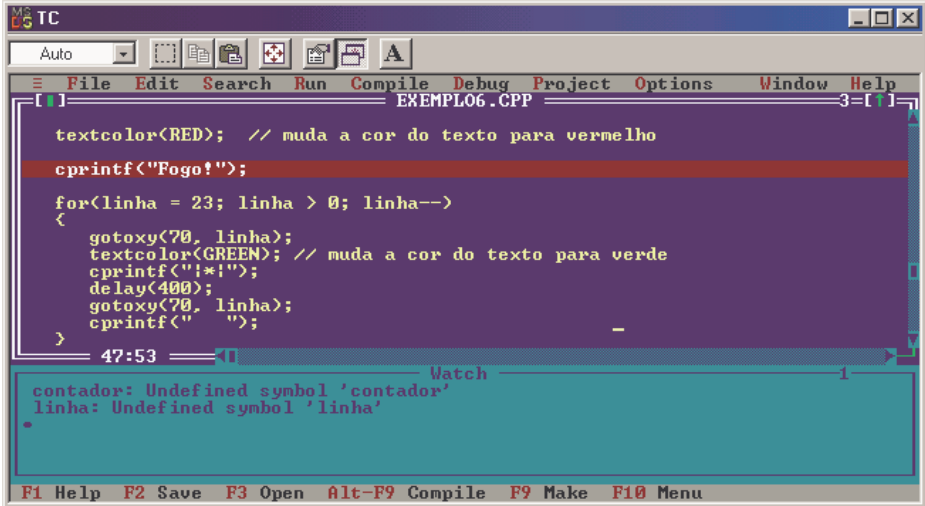


15.2 Adicionando pontos de parada (breakpoints)

Pode ser que você não queira executar seu programa passo-a-passo, mas sim, executa-lo até determinado ponto e pará-lo (para a partir daí, executa-lo passo-a-passo ou para identificar se ele entrou em um desvio condicional não desejado [if], por exemplo).

Para colocar um ponto de parada (*breakpoint*) no seu programa, vá com o cursor até a linha onde você deseja coloca-lo e pressione CTRL-F8. O Turbo-C coloca um ponto de parada na linha indicada e marca-a de vermelho:

Ponto de
parada
→



The screenshot shows the Turbo C++ IDE window titled 'TC'. The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The toolbar contains icons for Auto, Run, Compile, and other functions. The main editor window displays the code for 'EXEMPL06.CPP'. The code is as follows:

```
textcolor(RED); // muda a cor do texto para vermelho
cprintf("Fogo!");
for(linha = 23; linha > 0; linha-->
{
    gotoxy(70, linha);
    textcolor(GREEN); // muda a cor do texto para verde
    cprintf("!*");
    delay(400);
    gotoxy(70, linha);
    cprintf(" ");
}
```

The line `cprintf("Fogo!");` is highlighted in red, indicating a breakpoint. The status bar at the bottom shows '47:53' and a 'Watch' window with the following content:

```
contador: Undefined symbol 'contador'
linha: Undefined symbol 'linha'
```

The status bar also includes function key shortcuts: F1 Help, F2 Save, F3 Open, Alt-F9 Compile, F9 Make, and F10 Menu.

16 APÊNDICE III: TABELA ASCII

A tabela ASCII, acrônimo de American Standard Code for Information Interchange (Código Americano Padrão para o Intercambio de Informações), é um conjunto de valores que representam caracteres e códigos de controle armazenados ou utilizados em computadores. Nesta tabela, cada caractere (letra ou número) possui um código correspondente. Cada código ocupa 1 byte (tipo *char*), o que nos dá 255 posições.

Destas 255 posições, as primeiras 32 (0 a 31) correspondem a códigos de controle que são utilizados para controlar dispositivos (tais como monitores e impressoras). Logo, a maioria destes códigos não produz caracteres quando impressos em algum dispositivo (como a tela do computador). Da posição 32 até a 127 estão alocados os caracteres do conjunto padrão, correspondendo a caracteres do alfabeto latino (sem acentos, maiúsculos e minúsculos), dígitos (0 a 9) e alguns outros símbolos comuns. Os demais códigos (128 a 255) formam o conjunto estendido, e podem variar de região para região.

A seguir é apresentada a tabela ASCII padrão (código 0 a 127), que você encontra na maioria dos computadores.

Cód	Caractere	Cód	Caractere
0	NULL (nulo)	1	SOH (Start of Heading / Início de cabeçalho)
2	STX (Start of TeXt / Início de Texto)	3	ETX (End of TeXt / fim de texto)
4	EOT (End Of Transmission / fim de transmissão)	5	ENQ (ENQuiry / inquirição, consulta)
6	ACK (ACKnowledge / confirmação, entendido)	7	BEL (BELL, BEEP / Campanha)
8	BS (Backspace / retorno de 1 caractere)	9	HT (Horizontal Tab / Tabulação horizontal)
10	LF (Line Feed / alimentação, mudança de linha)	11	VT (Vertical Tab / Tabulação vertical)
12	FF (Form Feed / Alimentação de formulário)	13	CR (Carriage Return / retorno ao início da linha)
14	SO (Serial Out / Saída Serial) (Shift Out / deslocamento para fora)	15	SI (Serial In / Entrada Serial) (Shift In / deslocamento para dentro)
16	DLE (Data Link Escape / escape de conexão)	17	DC1/XON (Device Control1/controlado de dispositivo1)
18	DC2 (Device Control 2 / controle de dispositivo2)	19	DC3/XOFF (Device Control3/controlado de dispositivo3)
20	DC4 (Device Control 4 / controle de dispositivo4)	21	NAK (Negative AcKnowledge / confirmação negativa)
22	SYN (SYNchronous Idle / espera síncrona)	23	ETB (End Transm. Block/bloco de fim de transmissão)
24	CAN (Cancel / cancelamento)	25	EM (End of Media / Fim do meio ou mídia)
26	SUB (SUBstitute, substituir)	27	ESC (ESCape / escape)
28	FS (File Separator / Separador de arquivo)	29	GS (Group Separator / separador de grupo)
30	RS (Request to Send, Record Separator / requisição de envio, separador de registro)	31	US (Unit Separator / separador de unidade)

Códigos de controle

Conjunto padrão	Cód	Carac	Cód	Carac	Cód	Carac	Cód	Carac	Cód	Carac	Cód	Carac
	32	<espaço>	33	!	34	"	35	#	36	\$	37	%
	38	&	39	'	40	(41)	42	*	43	+
	44	,	45	-	46	.	47	/	48	0	49	1
	50	2	51	3	52	4	53	5	54	6	55	7
	56	8	57	9	58	:	59	;	60	<	61	=
	62	>	63	?	64	@	65	A	66	B	67	C
	68	D	69	E	70	F	71	G	72	H	73	I
	74	J	75	K	76	L	77	M	78	N	79	O
	80	P	81	Q	82	R	83	S	84	T	85	U
	86	V	87	W	88	X	89	Y	90	Z	91	[
	92	\	93]	94	^	95	_	96	`	97	a
	98	b	99	c	100	d	101	e	102	f	103	g
	104	h	105	i	106	j	107	k	108	l	109	m
	110	n	111	o	112	p	113	q	114	r	115	s
	116	t	117	u	118	v	119	w	120	x	121	y
122	z	123	{	124		125	}	126	~	127	<delete>	

Os códigos seguintes não são padrão e podem variar dependendo da configuração do seu computador, principalmente se você não estiver trabalhando com o MSDOS configurado para o idioma português ou estiver utilizando outro sistema operacional. Nestes casos, procure informar-se sobre o padrão UNICODE (<http://www.unicode.org>), utilizado por sistemas como o Windows, ou utilize o código-exemplo seguinte (após a tabela) que imprime a tabela ASCII na tela do seu computador.

Conjunto estendido (padrão brasileiro/português)	Cód	Carac	Cód	Carac	Cód	Carac	Cód	Carac	Cód	Carac	Cód	Carac
	128	Ç	129	ü	130	é	131	â	132	ä	133	à
	134	â	135	ç	136	ê	137	ë	138	è	139	ï
	140	î	141	ì	142	Ä	143	Å	144	É	145	æ
	146	Æ	147	ô	148	ö	149	ò	150	û	151	ù
	152	ÿ	153	Ö	154	Ü	155	ø	156	£	157	Ø
	158	×	159	f	160	á	161	í	162	ó	163	ú
	164	ñ	165	Ñ	166	ª	167	º	168	¿	169	®
	170	¬	171	½	172	¼	173	¡	174	«	175	»
	176	⋮	177	⋮	178	⋮	179		180	¡	181	Á
	182	Â	183	À	184	©	185	¶	186	¶	187	¶
	188	¶	189	¢	190	¥	191	¬	192	⌞	193	⌞
	194	⌞	195	⌞	196	—	197	†	198	ã	199	Ã
	200	⌞	201	¶	202	⌞	203	¶	204	¶	205	≡
	206	¶	207	π	208	δ	209	Ð	210	Ê	211	Ë
	212	È	213	¬	214	Í	215	Î	216	Ï	217	⌞
	218	Γ	219	■	220	■	221	¡	222	Ì	223	■
	224	Ó	225	β	226	Ô	227	Ò	228	õ	229	Õ
	230	Μ	231	Ɔ	232	Ɔ	233	Ú	234	Û	235	Û
	236	ý	237	Ý	238	-	239	´	240	-	241	±
	242	≡	243	¾	244	¶	245	§	246	÷	247	,
	248	°	249	“	250	.	251	¡	252	³	253	²
	254	■	255									

16.1 Programa exemplo que imprime a tabela ASCII na tela do seu computador

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
    int codigo;
    clrscr(); // limpa a tela

    for(codigo = 32; codigo < 255; codigo++) // começa no 32 para não mostrar
        printf("%3d: %-3c", codigo, codigo); // os códigos de controle

    getch();
}
```

17 APÊNDICE IV: RESUMO GERAL

Funções básicas da linguagem C:

Biblioteca	Funções
stdio.h	printf() Imprime strings e números na saída de dados padrão (tela). É possível especificar o formato de saída dos dados.
	putchar() Coloca (imprime) um caracter na saída de dados padrão (tela).
	puts() Imprime uma string na tela e passa para a próxima linha
	gets() Lê uma string do teclado
	scanf() Lê uma string ou número do teclado e coloca o valor lido em uma variável (no formato especificado).
conio.h	clrscr() Limpa a tela.
	getch() Espera que o usuário pressione uma tecla no teclado. Não mostra o caractere digitado.
	getche() Espera que o usuário pressione uma tecla no teclado. Mostra o caractere digitado.
	clreol() Apaga da posição corrente até o final da linha
	gotoxy() Posiciona o cursor na coluna e linha especificadas.
	textcolor() Muda a cor do texto.
	textbackground() Muda a cor do fundo do texto.
	cprintf() O mesmo que <i>printf</i> , porém, imprime utilizando o formato de cores especificado.

Algumas bibliotecas de funções da linguagem C:

math.h: funções matemáticas (seno, co-seno, tangente, raiz-quadrada, potenciação...).

graphics.h: funções de manipulação gráfica (linhas, barras, círculos, pontos...).

string.h: funções de manipulação de strings (vetores de caracteres).

File.h: funções de manipulação de arquivos.

Tipos básicos de variáveis da linguagem C:

números inteiros	int	inteiros entre -32.768 e +32.767
	unsigned int	inteiros entre 0 e 65.535
	long	inteiros entre -2.147.483.648 e +2.147.483.648
	unsigned long	inteiros entre 0 e 4.294.967.296
números de ponto flutuante	float	ponto flutuante entre 3,4e-38 a 3,4e+38
	double	ponto flutuante entre 1,7e-308 a 1,7e+308
	long double	ponto flutuante entre 3,4e-4932 a 1,1e+4932
Caracteres	char	caracteres ou números entre -128 a +127
	unsigned char	caracteres ou números entre 0 e 255

OBS: As faixas de valores podem mudar, dependendo do compilador, processador ou sistema operacional.

Principais operadores da linguagem C:

=	Atribuição
*	Multiplicação
/	Divisão
%	Resto (números inteiros)
+	Soma
-	Subtração
= =	(teste de) Igualdade
! =	(teste de) Diferença
&&	AND (“E” lógico)
	OR (“OU” lógico)
!	NOT (“Não” lógico, negação)
&	AND bit-a-bit
	OR bit-a-bit
++	Incremento
--	Decremento
&variável	Retorna o endereço de uma variável
*variável	Retorna o conteúdo de uma variável do tipo <i>ponteiro</i>

18 LEITURA RECOMENDADA

- Coplien, James. **Advanced C++ programming Styles and Idioms**. Ed. Addison Wesley.
- Schildt, Herbert,. **C completo e total**. Ed. Makron books.
- Mizrahi, Victorine Viviane. **Treinamento em linguagem C**. Ed. Makron books.