MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS PATO BRANCO
CURSO SUPERIOR DE TECNOLOGIA EM MANUTENÇÃO INDUSTRIAL

IA34M - INFORMÁTICA APLICADA - LINGUAGEM C -

PROF. CARLOS MARCELO DE OLIVEIRA STEIN

FEVEREIRO, 2006

1 INTRODUÇÃO

Neste material são passados os conceitos básicos da linguagem de programação C, que tem se tornado cada dia mais popular, devido à sua versatilidade e ao seu poder. Uma das grandes vantagens do C é que ele possui tanto características de "alto nível" quanto de "baixo nível". O C é uma linguagem de programação genérica que é utilizada para a criação de programas diversos como processadores de texto, planilhas eletrônicas, sistemas operacionais, programas de comunicação, programas para a automação industrial, gerenciadores de bancos de dados, programas de projeto assistido por computador, programas para a solução de problemas da Engenharia, Física, Química e outras Ciências, etc. Estudaremos a estrutura do ANSI C, o C padronizado pela ANSI. Sugere-se que o aluno realmente use o máximo possível dos exemplos, problemas e exercícios apresentados, gerando os programas executáveis com o seu compilador. Quando utilizamos o compilador aprendemos a lidar com mensagens de aviso, mensagens de erro, *bugs*, etc. Apenas ler os exemplos não basta. O conhecimento de uma linguagem de programação transcende o conhecimento de estruturas e funções. O C exige, além do domínio da linguagem em si, uma familiaridade com o compilador e experiência em achar *bugs* nos programas. É importante então que o aluno digite, compile e execute os exemplos apresentados.

2 LINGUAGEM C – PRIMEIROS PASSOS

2.1 O C é "Case Sensitive"

Vamos começar ressaltando um ponto de suma importância: o C é "*Case Sensitive*", isto é, *maiúsculas e minúsculas fazem diferença*. Se uma variável for declarada com o nome **var**, ela será diferente de **Var**, **VAR**, **VaR** ou **vAr**. Da mesma maneira, os comandos do C devem ser escritos da forma correta para o compilador interpretá-los como comandos.

2.2 O Primeiro Programa

Vejamos um primeiro programa em C:

```
#include <stdio.h>
int main () /* Um Primeiro Programa */
{
   printf ("Informática Aplicada – Primeiro Programa\n");
   return(0);
}
```

Compilando e executando este programa você verá que ele coloca a mensagem *Informática Aplicada – Primeiro Programa* na tela.

Como compilar programas usando o DevC++? Uma vez instalado o compilador, você pode criar um novo programa através da opção: *Arquivo* / *Novo* / *Arquivo Fonte*, ou utilizando as teclas de atalho *Ctrl+N*, ou ainda, utilizando o botão correspondente na barra de ferramentas. Na janela que aparece, você digita o seu programa. Veja que o DevC++ oferece para você um programa inicial que você pode apagar, se quiser. Uma vez digitado o seu programa, você pode compilá-lo e executá-lo através da opção: *Executar* / *Compilar & Executar*, ou pressionando diretamente *F9*, ou utilizando o botão *Compilar & Executar* na barra de ferramentas. Note que se você fizer isto para um programa que não espera nenhuma entrada do usuário, como o exemplo anterior, o programa será executado, terminará sua execução, e a janela onde ele está executando será automaticamente fechada. Com isto, você não conseguirá ver a saída do programa. Para evitar que isto aconteça, você pode modificar seus programas incluindo as linhas:

```
#include <stdlib.h>     no início do programa
system("pause");  antes do final do programa
```

Fazendo-se estas alterações no primeiro programa, tem-se:

```
#include <stdio.h>
#include <stdlib.h>
int main () /* Um Primeiro Programa */
{
   printf ("Informática Aplicada – Primeiro Programa\n");
   system("pause");
   return(0);
}
```

2.2.1 *Analisando o programa por partes*

A linha **#include <stdio.h>** diz ao compilador que ele deve incluir o arquivo de biblioteca de comandos **stdio.h**. Neste arquivo existem declarações de funções úteis para entrada e saída de dados (std = standard, padrão em inglês; io = Input/Output, entrada e saída ==> stdio = Entrada e saída padronizadas). Toda vez que você quiser usar uma destas funções deve-se incluir este comando. O C possui diversas bibliotecas de comandos.

Quando fazemos um programa, é uma boa idéia usar comentários que ajudem a elucidar o funcionamento do mesmo. No caso acima temos um comentário: **/* Um Primeiro Programa */**. O compilador C desconsidera qualquer coisa que esteja começando com **/*** e terminando com ***/**.

A linha **int main()** define uma função de nome **main**. Todos os programas em C têm que ter uma função **main**, pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves **{ }**. O código que estiver dentro das chaves será executado sequencialmente quando a função for chamada. A palavra **int** indica que esta função retorna um número inteiro.

A única coisa que este programa *realmente* faz é chamar a função **printf()**, passando o texto **"Informática Aplicada – Primeiro Programa\n"** como argumento. É por causa do uso da função **printf()** que a biblioteca **stdio.h** foi incluída. A função **printf()** neste caso irá apenas colocar o texto na tela do computador. A **\n** é uma constante chamada de *constante barra invertida de nova linha* que é interpretado como um comando de mudança de linha, isto é, após imprimir *Informática Aplicada – Primeiro Programa* o cursor passará para a próxima linha. É importante observar também que os *comandos* do C terminam com **;**.

2.3 *Comentários*

O compilador ignora os comentários existentes no código fonte, ou seja, eles não interferem no programa executável. O uso de comentários torna o código do programa mais fácil de se entender. Existem basicamente duas formas de inserir comentários: com o comando // e com o comando **/* */**. O primeiro é utilizado para comentar o restante da linha (após //), e o segundo para comentar tudo entre **/*** e ***/**. Um comentário pode, desta forma, ter mais de uma linha.

2.4 *Nomes de Variáveis*

As variáveis no C podem ter qualquer nome se algumas condições forem satisfeitas: o nome deve começar com uma letra ou sublinhado (_) e os caracteres subsequentes podem ser letras, números ou sublinhado. Além disso, o nome de uma variável não pode ser igual a uma palavra reservada, nem igual ao nome de uma função declarada pelo programador, ou pelas bibliotecas do C. Variáveis de até 32 caracteres são aceitas. Mais uma coisa: é bom sempre lembrar que o C é "case sensitive" e, portanto deve-se prestar atenção às maiúsculas e minúsculas.

2.5 *Palavras Reservadas do C*

Todas as linguagens de programação têm palavras reservadas. As palavras reservadas não podem ser usadas a não ser para seus propósitos originais. Como o C é "case sensitive" podemos declarar uma variável **For**, apesar de haver uma palavra reservada **for**, mas isto não é uma coisa

recomendável de se fazer, pois pode gerar confusão. Apresentamos a seguir as palavras reservadas do ANSI C.

auto	*default*	*float*	*register*	*struct*	*volatile*
break	*do*	*for*	*return*	*switch*	*while*
case	*double*	*goto*	*short*	*typedef*	
char	*else*	*if*	*signed*	*union*	
const	*enum*	*int*	*sizeof*	*unsigned*	
continue	*extern*	*long*	*static*	*void*	

2.6 *Os Tipos do C*

O C tem tipos básicos: **char** (um caracter), **int** (um número inteiro), **float** (um número com ponto flutuante) e **double** (ponto flutuante de precisão dupla). Para cada um dos tipos de variáveis existem os modificadores de tipo. Os modificadores de tipo do C são quatro: **signed**, **unsigned**, **long** e **short**. Ao **float** não se pode aplicar nenhum e ao **double** pode-se aplicar apenas o **long**. Os quatro podem ser aplicados a inteiros. A intenção é que **short** e **long** devam prover tamanhos diferentes de inteiros onde isto for prático. **int** normalmente terá o tamanho natural para uma determinada máquina. Assim, em uma máquina de 16 bits, **int** provavelmente terá 16 bits e em uma máquina de 32, 32 bits. Na verdade, cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que **short** e **int** devem ocupar pelo menos 16 bits, **long** pelo menos 32 bits, e **short** não pode ser maior que **int**, que não pode ser maior que **long**. A seguir estão listados os tipos de dados permitidos e seus valores máximos e mínimos em um compilador típico para um hardware de 16 bits:

Tipo	Número de bits	Intervalo	
		Inicio	Fim
char	8	-128	127
unsigned char	8	0	255
signed char	8	-128	127
int	16	-32.768	32.767
unsigned int	16	0	65.535
signed int	16	-32.768	32.767
short int	16	-32.768	32.767
unsigned short int	16	0	65.535
signed short int	16	-32.768	32.767
long int	32	-2.147.483.648	2.147.483.647
signed long int	32	-2.147.483.648	2.147.483.647
unsigned long int	32	0	4.294.967.295
float	32	3,4E-38	3.4E+38
double	64	1,7E-308	1,7E+308
long double	80	3,4E-4932	3,4E+4932

O tipo **long double** é o tipo de ponto flutuante com maior precisão. É importante observar que os intervalos de ponto flutuante, na tabela acima, estão indicados em faixa de *expoente*, mas os números podem assumir valores tanto positivos quanto negativos.

2.7 *Declaração e Inicialização de Variáveis*

As variáveis no C devem ser declaradas antes de serem usadas. A forma geral da declaração de variáveis é:

```
tipo_da_variável lista_de_variáveis;
```

As variáveis da lista de variáveis terão todas o mesmo tipo e deverão ser separadas por vírgula. Como o tipo *default* (padrão) do C é o **int**, quando vamos declarar variáveis **int** com algum dos modificadores de tipo, basta colocar o nome do modificador de tipo. Assim um **long** basta para declarar um **long int**. Por exemplo, as declarações

```
char ch, letra;
long cont;
```

```
float pi;
```

declaram duas variáveis do tipo **char** (ch e letra), uma variável **long int** (cont) e um **float** (pi).

Podemos inicializar variáveis no momento de sua declaração. Isto é importante, pois quando o C cria uma variável ele *não* a inicializa. Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor *indefinido* e que não pode ser utilizado para nada. *Nunca* presuma que uma variável declarada vale zero ou qualquer outro valor. Exemplos de inicialização são dados abaixo:

```
char ch='D';
int cont=0;
float pi=3.141;
```

2.8 A Função printf

A função **printf()** tem a seguinte forma geral:

```
printf (string_de_controle, lista_de_argumentos);
```

Teremos, na *string de controle*, uma descrição de tudo que a função vai colocar na tela. A string de controle mostra não apenas os caracteres que devem ser colocados na tela, mas também quais as variáveis e suas respectivas posições. Isto é feito usando-se os códigos de controle, que usam a notação **%**. Na string de controle indicamos quais, de qual tipo e em que posições estão as variáveis a serem apresentadas. É muito importante que, para cada código de controle, tenhamos um argumento na lista de argumentos. Apresentamos agora alguns dos códigos **%**:

Código	Significado
%d	Inteiro
%f	Float
%c	Caractere
%s	String
%%	Coloca na tela um %

Vamos ver alguns exemplos de **printf()** e o que eles exibem:

```
printf ("Teste %% %%") -> "Teste % %"
printf ("%f",40.345) -> "40.345"
printf ("%c = %d",'D',120) -> "D = 120"
printf ("Veja %s exemplo","este") -> "Veja este exemplo"
printf ("%s%d%%","Juros de ",10) -> "Juros de 10%"
```

2.8.1 Exercício:

Escreva um programa que declare seis variáveis inteiras e atribua os valores 10, 20, 30, ..., 60 a elas. Declare 6 variáveis caracteres e atribua a elas as letras c, o, e, l, h, o. Finalmente, o programa deverá imprimir, usando todas as variáveis declaradas:

As variáveis inteiras contêm os números: 10,20,30,40,50,60

O animal contido nas variáveis caracteres é o coelho

2.8.2 Constantes de barra invertida

Para nos facilitar a tarefa de programar, o C utiliza vários códigos chamados códigos de barra invertida. A lista completa dos códigos de barra invertida é dada a seguir:

Código	Significado	Código	Significado
\"	Aspas	\f	Alimentação de formulário ("form feed")
\'	Apóstrofo	\n	Nova linha ("new line")
\\	Barra invertida	\t	Tabulação horizontal ("tab")
\0	Nulo (0 em decimal)	\v	Tabulação vertical
\a	Sinal sonoro ("beep")	\N	Constante octal (N é o valor da constante)
\b	Retrocesso ("back")	\xN	Constante hexadecimal (N é o valor da constante)

2.9 A Função scanf

O formato geral da função **scanf()** é:

```
scanf (string_de_controle, lista_de_argumentos);
```

Usando a função **scanf()** podemos pedir dados ao usuário. Mais uma vez, devemos ficar atentos a fim de colocar o mesmo número de argumentos que o de códigos de controle na string de controle. Outra coisa importante é lembrarmos de colocar o **&** antes das variáveis da lista de argumentos.

2.10 O Segundo Programa

Podemos agora tentar um programa mais complicado:

```
#include <stdlib.h>
#include <stdio.h>
int main ()
{
  int Minutos; // Declaração de Variável
  float Horas;
  printf("Entre com os minutos: "); // Entrada de Dados
  scanf("%d",&Minutos);
  Horas=Minutos/60.0; // Conversão Minutos->Horas
  printf("\n %d minutos equivalem a %f horas.\n",Minutos,Horas);
  system("pause");
  return(0);
}
```

Vamos entender como este programa funciona. Inicialmente são declaradas duas variáveis: uma inteira (**Minutos**) e uma ponto flutuante (**Horas**). É feita então uma chamada à função **printf()**, que coloca uma mensagem na tela.

Queremos agora ler um dado que será fornecido pelo usuário e colocá-lo na variável **Minutos**. Para isso usamos a função **scanf()**. A string **"%d"** diz à função que iremos ler um inteiro. O segundo parâmetro passado à função diz em que variável deverá ser armazenado o dado lido. É importante ressaltar a necessidade de se colocar um **&** antes do nome da variável a ser lida quando se usa a função **scanf()**. Observe que, no C, quando temos mais de um parâmetro para uma função, eles serão separados por vírgula.

Temos então uma expressão matemática simples que atribui a **Horas** o valor de **Minutos** dividido por 60.0. Como Horas é uma variável float, o compilador fará uma conversão entre os tipos das variáveis (o número 60.0 está com este formato para forçar esta conversão).

A segunda chamada à função **printf()** tem três argumentos. A string **"\n %d minutos equivalem a %f horas.\n"** diz à função para dar uma nova linha (passar para a próxima linha), colocar um inteiro (Minutos) na tela, colocar a mensagem **" minutos equivalem a "**, colocar um ponto flutuante (Horas) na tela, colocar a mensagem **" horas."** e dar mais uma nova linha. Os outros parâmetros são as variáveis das quais devem ser lidos os valores.

2.10.1 Exercício:

O que faz o seguinte programa?

```
#include <stdlib.h>
#include <stdio.h>
main()
{
  int x;
  scanf("%d",&x);
  printf("%d",x);
  system("pause");
}
```

2.11 Caracteres

Os caracteres são um tipo de dado: o **char**. O C trata os caracteres como sendo variáveis de um *byte* (8 *bits*). Um *bit* é a menor unidade de armazenamento de informações em um computador. Na linguagem C, também podemos usar um **char** para armazenar valores numéricos inteiros, além de usá-lo para armazenar caracteres de texto. Para indicar um caractere de texto usamos apóstrofos. Veja um exemplo de programa que usa caracteres:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      char Ch;
      Ch='D';
      printf ("%c",Ch);
      system("pause");
      return(0);
}
```

No programa acima, **%c** indica que **printf()** deve colocar um caractere na tela. Como dito anteriormente, um **char** também é usado para armazenar um número inteiro. Este número é conhecido como o código ASCII correspondente ao caractere. Veja o programa abaixo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  char Ch;
  Ch='D';
  printf ("%d",Ch); /* Imprime o caracter como inteiro */
  system("pause");
  return(0);
}
```

Este programa vai imprimir o número 68 na tela, que é o código ASCII correspondente ao caractere 'D' (d maiúsculo).

2.11.1 Exercício:

Escreva um programa que leia um caractere digitado pelo usuário, imprima o caractere digitado e o código ASCII correspondente a este caractere.

2.12 As Funções getch e getche

Muitas vezes queremos ler um caractere fornecido pelo usuário. Para isto pode-se utilizar as funções **getch()** ou **getche()**. Ambas retornam o caractere pressionado, sendo que **getche()** imprime o caractere na tela antes de retorná-lo e **getch()** apenas retorna o caractere pressionado sem imprimi-lo na tela. Ambas as funções podem ser encontradas na biblioteca **conio.h**. Eis um exemplo que usa a função **getch()**:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main ()
{
      char Ch;
      Ch=getch();
      printf ("Voce pressionou a tecla %c\n",Ch);
      system("pause");
      return(0);
}
```

Equivalente para o programa acima, sem usar **getch()**:

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



```
#include <stdlib.h>
int main ()
{
      char Ch;
      scanf("%c", &Ch);
      printf ("Voce pressionou a tecla %c\n",Ch);
      system("pause");
      return(0);
}
```

A principal diferença da versão que utiliza **getch()** para a versão que não utiliza **getch()** é que no primeiro caso o usuário simplesmente aperta a tecla e o sistema lê diretamente a tecla pressionada. No segundo caso, é necessário apertar também a tecla <ENTER>. Lembre-se que, se você quiser manter a portabilidade de seus programas, não deve utilizar as funções **getch** e **getche**, pois estas não fazem parte do padrão ANSI C.

2.13 Strings

No C uma *string* é um vetor de caracteres terminado com um caractere nulo. O caracter nulo é um caractere com valor inteiro igual a zero (código ASCII igual a 0). O caractere nulo também pode ser escrito usando a convenção de barra invertida do C como sendo **\0**. Para declarar uma string podemos usar o seguinte formato geral:

```
char nome_da_string[tamanho];
```

Isto declara um vetor de caracteres (uma *string*) com número de posições igual a *tamanho*. Note que, como temos que reservar um espaço para o caractere nulo, temos que declarar o comprimento da string como sendo, no mínimo, um caractere maior que a maior string que pretendemos armazenar. Vamos supor que declaremos uma string de 8 posições e coloquemos a palavra Texto nela. Teremos:

T	e	x	t	o	\0	...	...

No caso acima, as duas células não usadas têm valores indeterminados. Isto acontece porque o C *não* inicializa variáveis, cabendo ao programador esta tarefa. Se quisermos ler uma string fornecida pelo usuário podemos usar a função **gets()**. Um exemplo do uso desta função é apresentado abaixo. A função **gets()** coloca o terminador nulo na string, quando você aperta a tecla "Enter".

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
 char string[100];
 printf ("Digite uma string: ");
 gets (string);
 printf ("\n\nVoce digitou %s.\n",string);
 system("pause");
 return(0);
}
```

Neste programa, o tamanho máximo da string que você pode entrar é de 99 caracteres. Se você entrar com uma string de comprimento maior, o programa irá aceitar, mas os resultados podem ser desastrosos.

Como as strings são vetores de caracteres, para se acessar um determinado caracter de uma string, basta usar um índice para acessar o caracter desejado. Suponha uma string chamada *str*. Podemos acessar a *segunda* letra de *str* da seguinte forma:

```
str[1] = 'a';
```

Observe que na linguagem C o índice ***começa em zero.*** Assim, a primeira letra da string sempre estará na posição 0. A segunda letra sempre estará na posição 1 e assim sucessivamente.

Segue um exemplo que imprimirá a terceira letra da string "Texto", apresentada acima. Em seguida, ele mudará esta letra e apresentará a nova string no final.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 char str[10] = "Texto";
 printf("\n\nString: %s", str);
 printf("\nTerceira letra: %c", str[2]);
 str[2] = 'n';
 printf("\nAgora a terceira letra eh: %c", str[2]);
 printf("\n\nString resultante: %s\n", str);
 system("pause");
 return(0);
}
```

Nesta string, o terminador nulo está na posição 5. Das posições 0 a 5, sabemos que temos caracteres válidos, e portanto podemos escrevê-los. Note a forma como inicializamos a string **str** com os caracteres 'T' 'e' 'x' 't' 'o' e '\0' simplesmente declarando char str[10] = "Texto". Veremos, posteriormente que "Texto" (uma cadeia de caracteres entre aspas) é o que chamamos de string constante, isto é, uma cadeia de caracteres que está pré-carregada com valores que não podem ser modificados. Já a string str é uma string variável, pois podemos modificar o que nela está armazenado, como de fato fizemos.

2.13.1 Exercício:

Escreva um programa que leia duas strings e as coloque na tela. Imprima também a segunda letra de cada string.

2.14 Funções

Uma função é um bloco de código que pode ser usado diversas vezes na execução de um programa. O uso de funções permite que o programa fique mais legível, e melhor estruturado. Na verdade, um programa em C consiste de várias funções colocadas juntas. A forma geral de uma função é:

```
tipo_de_retorno nome_da_função (lista_de_argumentos)
{
 código_da_função
}
```

Abaixo o tipo mais simples de função:

```
#include <stdio.h>
#include <stdlib.h>
void mensagem ()  // Função simples: só imprime o texto
{
   printf ("Informática ");
}

main ()
{
   mensagem();
   printf ("Aplicada\n");
   system("pause");
}
```

Este programa terá o mesmo resultado que o primeiro exemplo visto. O que ele faz é definir uma função **mensagem()** que coloca uma string na tela e não retorna nada (**void**). Depois esta função é chamada a partir de **main()** (que também é uma função).

2.14.1 Argumentos

Argumentos são as entradas que a função recebe. É através dos argumentos que passamos *parâmetros* para a função. Já vimos funções com argumentos. As funções **printf()** e **scanf()** são funções que recebem argumentos. Vamos ver um exemplo simples de função com argumentos:

```
#include <stdio.h>
#include <stdlib.h>
void quad(int x)  // Calcula o quadrado de x
{
   printf ("O quadrado e %d",(x*x));
}

main ()
{
   int num;
   printf ("Entre com um numero: ");
   scanf ("%d",&num);
   printf ("\n\n");
   quad(num);
   system("pause");
}
```

Na definição de **quad()** dizemos que a função receberá um argumento inteiro **x**. Quando fazemos a chamada à função, o inteiro **num** é passado como argumento. Há alguns pontos a observar. Em primeiro lugar temos de satisfazer aos requisitos da função quanto ao tipo e a quantidade de argumentos quando a chamamos. Apesar de existirem algumas conversões de tipo que o C faz automaticamente, é importante ficar atento. Em segundo lugar, não é importante o nome da variável que se passa como argumento, ou seja, a variável **num**, ao ser passada como argumento para **quad()** é copiada para a variável **x**. Dentro de **quad()** trabalha-se apenas com **x**. Se mudarmos o valor de **x** dentro de **quad()** o valor de **num** na função **main()** permanece inalterado.

Vamos ver um exemplo de função com mais de uma variável. Repare que, neste caso, os argumentos são separados por vírgula e que deve ser explicitado o tipo de cada um dos argumentos, um a um. Note também que não é necessário que todos os argumentos passados para a função sejam variáveis, porque serão copiados para a variável de entrada da função.

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

void mult (float a, float b, float c)    // Multiplica 3 números
{
   printf ("%f",a*b*c);
}

main ()
{
   float x,y;
   x=23.5;
   y=12.9;
   mult (x,y,3.87);
   system("pause");
}
```

2.14.2 Retornando valores

Muitas vezes é necessário fazer com que uma função retorne um valor. As funções que vimos até aqui não retornam nada, pois especificamos um retorno **void**. Podemos especificar um tipo de retorno indicando-o antes do nome da função. Mas para dizer ao C *o que* vamos retornar precisamos da palavra reservada **return**. Assim fica fácil fazer uma função para multiplicar dois inteiros e que retorna o resultado da multiplicação. Veja:

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

```
#include <stdlib.h>
int prod (int x,int y)
{
  return (x*y);
}

main ()
{
  int saida;
  saida=prod (12,7);
  printf ("A saida e: %d\n",saida);
  system("pause");
}
```

Veja que, como prod retorna o valor de 12 multiplicado por 7, este valor pode ser usado em uma expressão qualquer. No programa fizemos a atribuição deste resultado à variável saida, que posteriormente foi impressa usando **printf**. Uma observação adicional: se não especificarmos o tipo de retorno de uma função, o compilador C automaticamente suporá que este tipo é inteiro. Porém, não é uma boa prática não se especificar o valor de retorno.

Mais um exemplo de função, que agora recebe dois **float**s e também retorna um **float**. Repare que neste exemplo especificamos um valor de retorno para a função **main** (**int**) e retornamos zero. Normalmente é isto que fazemos com a função **main**, que retorna zero quando ela é executada sem qualquer tipo de erro:

```
#include <stdio.h>
#include <stdlib.h>
float prod (float x, float y)
{
  return (x*y);
}

int main ()
{
  float saida;
  saida=prod (45.2,0.0067);
  printf ("A saida e: %f\n",saida);
  system("pause");
  return(0);
}
```

2.14.3 Exercício:

a) Escreva uma função que some dois inteiros e retorne o valor da soma.

3 ESTRUTURAS DE CONTROLE DE FLUXO

As estruturas de controle de fluxo são fundamentais para qualquer linguagem de programação. Sem elas só haveria uma maneira do programa ser executado: de cima para baixo, comando por comando. Não haveria condições, repetições ou saltos. A linguagem C possui diversos comandos de controle de fluxo. É possível resolver todos os problemas sem utilizar todas elas, mas devemos nos lembrar que a elegância e facilidade de entendimento de um programa dependem do uso correto das estruturas no local certo.

3.1 O Comando if

O comando **if** representa uma tomada de decisão do tipo "SE isto ENTÃO aquilo". A sua forma geral é:

```
if (condição) declaração;
```

O comando **if** avalia a expressão da *condição*. A *declaração* será executada somente se a *condição* for satisfeita (for verdadeira). A *declaração* pode ser um bloco de código ou apenas um

comando. É interessante notar que, no caso da *declaração* ser um bloco de código, não é necessário (e nem permitido) o uso do **;** no final do bloco. Isto é uma regra geral para blocos de código. Abaixo apresentamos um exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
      if (num>10) printf ("\n\nO numero e maior que 10");
      if (num==10)
      {
            printf ("\n\nVoce acertou!\n");
            printf ("O numero e igual a 10.");
      }
      if (num<10) printf ("\n\nO numero e menor que 10");
      system("pause");
      return (0);
}
```

No programa acima a expressão **num>10** é avaliada e retorna um valor diferente de zero, se verdadeira, e zero, se falsa. Repare que quando queremos testar igualdades usamos o operador `==` e não `=`. Isto é porque o operador `=` representa *apenas* uma atribuição. Isto pode parecer estranho à primeira vista, mas se escrevêssemos

```
if (num=10) ... /* Isto está errado */
```

o compilador iria *atribuir* o valor 10 à variável **num** e a expressão **num=10** iria retornar 10, fazendo com que o nosso valor de **num** fosse adulterado e fazendo com que a declaração fosse executada sempre. Este problema gera erros freqüentes entre iniciantes e, portanto, muita atenção deve ser tomada.

Os operadores de comparação são:

Operador	Significado
==	igual
!=	diferente
>	maior que
<	menor que
>=	maior ou igual
<=	menor ou igual

3.1.1 O else

Podemos pensar no comando **else** como sendo um complemento do comando **if**. O comando **if** completo tem a seguinte forma geral:

```
if (condição) declaração_1;
else declaração_2;
```

A expressão da *condição* será avaliada. Se ela for satisfeita, a *declaração_1* será executada, caso contrário, a *declaração_2* será executada. É importante nunca esquecer que, quando usamos a estrutura **if-else**, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Abaixo está um exemplo do uso do **if-else** que deve funcionar como o programa da seção anterior.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
```

```
        if (num==10)
        {
              printf ("\n\nVoce acertou!\n");
              printf ("O numero e igual a 10.\n");
        }
        else
        {
              printf ("\n\nVoce errou!\n");
              printf ("O numero e diferente de 10.\n");
        }
        system("pause");
        return(0);
}
```

3.1.2 O if-else-if

A estrutura **if-else-if** é apenas uma extensão da estrutura **if-else**. Sua forma geral pode ser escrita como sendo:

```
if (condição_1) declaração_1;
else if (condição_2) declaração_2;
     else if (condição_3) declaração_3;
          . . .
          else if (condição_n) declaração_n;
               else declaração_default;
```

Esta estrutura funciona da seguinte maneira: o programa testa as condições até encontrar uma expressão que seja satisfeita. Neste caso ele executa a declaração correspondente. Só uma declaração será executada, ou seja, só será executada a declaração equivalente à *primeira* condição que for satisfeita. A última declaração (*default*) será executada no caso de nenhuma condição ser satisfeita, e é opcional. Um exemplo desta estrutura:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
      if (num>10)
            printf ("\n\nO numero e maior que 10");
      else if (num==10)
            {
               printf ("\n\nVoce acertou!\n");
               printf ("O numero e igual a 10.");
            }
            else if (num<10)
                     printf ("\n\nO numero e menor que 10");
      system("pause");
      return(0);
}
```

3.1.3 A expressão condicional

Quando o compilador avalia uma condição, ele quer um valor de retorno para poder tomar a decisão. Mas esta expressão não necessita ser uma expressão no sentido convencional. Uma variável sozinha pode ser uma *expressão* e esta retorna o seu próprio valor. Assim, as seguintes expressões:

```
int num;
if (num!=0) ....
if (num==0) ....
```

equivalem a

```
int num;
if (num) ....
```



```
if (!num) ....
```

3.1.4 ifs aninhados

O **if** aninhado é simplesmente um **if** dentro da declaração de um outro **if** externo. O único cuidado que devemos ter é o de saber exatamente a qual **if** um determinado **else** está ligado. Vejamos um exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
      if (num==10)
      {
            printf ("\n\nVoce acertou!\n");
            printf ("O numero e igual a 10.\n");
      }
      else
      {
            if (num>10)
            {
                  printf ("O numero e maior que 10.");
            }
            else
            {
                  printf ("O numero e menor que 10.");
            }
      }
      system("pause");
      return(0);
}
```

3.1.5 O Operador ?

De uma maneira geral, expressões do tipo:

```
if (condição)
   expressão_1;
else
   expressão_2;
```

podem ser simplificada usando-se o operador **?** da seguinte maneira:

```
condição?expressão_1:expressão_2;
```

Assim, uma expressão como:

```
if (a>0)
 b=-150;
else
 b=150;
```

pode ser simplificada por:

```
b=a>0?-150:150;
```

O operador **?** é limitado (não atende a uma gama muito grande de casos) mas pode ser usado para simplificar expressões complicadas. Uma aplicação interessante é a do contador circular. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int index = 0, contador;
```

```
        char letras[6] = "Texto";
        for (contador=0; contador < 1000; contador=contador+1)
        {
              printf("\n%c",letras[index]);
              index=(index==5)? index=0: index=index+1;
        }
system("pause");
}
```

A palavra *Texto* é escrita verticalmente na tela até a variável contador determinar o término do programa. Enquanto isto a variável index assume os valores 0, 1, 2, 3, 4, 0, 1, ... progressivamente.

3.1.6 Exercícios:

a) Altere o último exemplo para que ele escreva cada letra 5 vezes seguidas. Para isto, use um **if** para testar se o contador é divisível por cinco (utilize o operador **%**) e só então realizar a atualização em index.

3.2 O Comando switch

O comando **if-else** e o comando **switch** são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o **if**, mas o comando **switch** tem aplicações valiosas. O comando **switch** é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos. Sua forma geral é:

```
switch (variável)
{
case constante_1:
   declaração_1;
   break;
case constante_2:
   declaração_2;
   break;
. . .
case constante_n:
   declaração_n;
   break;
default
   declaração_default;
}
```

Podemos fazer uma analogia entre o **switch** e a estrutura **if-else-if** apresentada anteriormente. A diferença fundamental é que a estrutura **switch** *não* aceita expressões, aceita apenas constantes. O **switch** testa a variável e executa a declaração cujo **case** corresponda ao valor atual da variável. A declaração **default** é opcional e será executada apenas se a variável que está sendo testada não for igual a nenhuma das constantes.

O comando **break** faz com que o **switch** seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando **switch**. Se após a execução da declaração não houver um **break**, o programa continuará executando. Isto pode ser útil em algumas situações, mas recomenda-se cuidado. Veremos agora um exemplo do comando **switch**:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
      switch (num)
      {
            case 9: printf ("\n\nO numero e igual a 9.\n");
            break;
            case 10: printf ("\n\nO numero e igual a 10.\n");
```



```
            break;
            case 11: printf ("\n\nO numero e igual a 11.\n");
            break;
            default: printf ("\n\nNao e 9 nem 10 nem 11.\n");
      }
      system("pause");
      return(0);
}
```

3.3 O Comando for

O **for** é a primeira de uma série de três estruturas para se trabalhar com *loops* (laços) de repetição. As outras são **while** e **do**. As três compõem a segunda família de comandos de controle de fluxo. Podemos pensar nesta família como sendo a das estruturas de repetição controlada. O comando **for** é usado para repetir um comando (ou bloco de comandos) diversas vezes, de maneira que se possa ter um bom controle sobre o laço. Sua forma geral é:

```
for (inicialização;condição;incremento) declaração;
```

A declaração no comando **for** também pode ser um bloco ({ }) e neste caso o **;** é omitido. O melhor modo de se entender o **for** é ver de que maneira ele funciona. O **for** é equivalente a se fazer o seguinte:

```
inicialização;
if (condição)
{
  declaração;
  incremento;
  "Volte para o comando if"
}
```

Podemos ver, então, que o **for** executa a inicialização e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Abaixo vemos um programa que coloca os primeiros 100 números inteiros na tela:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int cont;
      for (cont=1;cont<=100;cont=cont+1) printf ("%d ",cont);
      system("pause");
      return(0);
}
```

Outro exemplo interessante é mostrado a seguir: o programa lê uma string e conta quantos dos caracteres desta string são iguais à letra 'c'

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
 char string[100]; /* String, ate' 99 caracteres */
 int i, cont;
 printf("\n\nDigite uma frase: ");
 gets(string); /* Le a string */
 printf("\n\nFrase digitada:\n%s", string);
 cont = 0;
 for (i=0; string[i] != '\0'; i=i+1)
 {
      if ( string[i] == 'c' )  /* Se for a letra 'c' */
      cont = cont +1;   /* Incrementa o contador */
 }
 printf("\nNumero de caracteres c = %d", cont);
 system("pause");
```

```
 return(0);
}
```

Note o teste que está sendo feito no **for**: o caractere armazenado em string[i] é comparado com '\0' (caractere final da string). Caso o caractere seja diferente de '\0', a condição é verdadeira e o bloco do **for** é executado. Dentro do bloco existe um **if** que testa se o caractere é igual a 'c'. Caso seja, o contador de caracteres c é incrementado.

3.3.1 Exercícios

a) Escreva um programa que coloque os números de 1 a 100 na tela na ordem inversa (começando em 100 e terminando em 1).

b) Escreva um programa que leia uma string, conte quantos caracteres desta string são iguais a 'a' e substitua os que forem iguais a 'a' por 'b'. O programa deve imprimir o número de caracteres modificados e a string modificada.

O **for** na linguagem C é bastante flexível. Temos acesso à inicialização, à condição e ao incremento. Um ponto importante é que podemos omitir qualquer um dos elementos do **for**, isto é, se não quisermos uma inicialização poderemos omiti-la. Isto nos permite fazer o que quisermos com o comando, conforme veremos a seguir.

3.3.2 O laço infinito

O laço infinito tem a forma

```
for (inicialização; ;incremento) declaração;
```

Este bloco de programa é um laço infinito porque será executado para sempre (não existindo a condição, ela será sempre considerada verdadeira), a não ser que ele seja interrompido. Para interromper um laço como este usamos o comando **break**. O comando **break** vai quebrar o laço infinito e o programa continuará sua execução normalmente. Como exemplo vamos ver um programa que faz a leitura de uma tecla e sua impressão na tela, até que o usuário aperte uma tecla sinalizadora de final (um FLAG). O nosso FLAG será a letra 'X'.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main ()
{
      int cont;
      char ch;
      for (cont=1; ;cont=cont+1)
      {
            ch = getch();
            if (ch == 'X') break;
            printf("\nLetra: %c",ch);
      }
      system("pause");
      return(0);
}
```

3.3.3 O laço sem conteúdo

Laço sem conteúdo é aquele no qual se omite a declaração. Sua forma geral é (atenção ao ponto e vírgula!):

```
for (inicialização;condição;incremento);
```

Uma das aplicações desta estrutura é gerar tempos de espera. O programa a seguir faz isto.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
```



```
{
      long int i;
      printf("\a"); // Imprime o caracter de alerta (um beep)
      for (i=0; i<1000000000; i=i+1);// Espera 1000000000 iteracoes
      printf("\a"); // Novo beep
      system("pause");
      return(0);
}
```

Um problema do código anterior, é a dependência do tempo de espera com a velocidade de processamento do computador. Para gerar um tempo de espera em milisegundos, pode-se utilizar o comando Sleep, da seguinte forma:

```
Sleep(tempo);
```

Para utilizar este comando, deve-se incluir a biblioteca windows. Observe que a primeira letra do comando é maiúscula, e que o tempo deve estar em milisegundos. Por exemplo, para que o programa espere 5 segundos:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h> // biblioteca onde está o comando Sleep
int main ()
{
      long int i;
      printf("\a"); // Imprime o caracter de alerta (um beep)
      Sleep(5000);// Espera 5 segundos
      printf("\a"); // Novo beep
      system("pause");
      return(0);
}
```

3.3.4 Exercícios

a) Faça um programa que inverta uma string: leia a string com **gets** e armazene-a invertida em outra string. Use o comando **for** para varrer a string até o seu final.

b) Escreva um programa, utilizando os comandos **for** e **switch**, que leia uma string (use **gets**) e substitua todos os espaços e tabulações (**\t**) por caracteres de nova linha (**\n**). O **for** deve ser encerrado quando o caracter de final da string (**\0**) for encontrado.

3.4 O Comando while

O comando **while** tem a seguinte forma geral:

```
while (condição) declaração;
```

Assim como fizemos para o comando **for**, vamos tentar mostrar como o **while** funciona fazendo uma analogia. Então o **while** seria equivalente a:

```
if (condição)
{
  declaração;
  "Volte para o comando if"
}
```

Podemos ver que a estrutura **while** testa uma condição. Se esta for verdadeira, a declaração é executada e faz-se o teste novamente, e assim por diante. Assim como no caso do **for**, podemos fazer um laço infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um laço sem conteúdo. No exemplo abaixo, o programa só irá finalizar quuando o usuário digitar a tecla 'q':

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main ()
```

```
{
      char Ch;
      Ch='\0';
      while (Ch!='q')
      {
            Ch = getch();
      }
      system("pause");
      return(0);
}
```

3.4.1 *Exercícios*

Refaça o programa anterior. Use o comando *while* para fechar o loop.

3.5 *O Comando do-while*

A terceira estrutura de repetição que veremos é o **do-while**. De forma geral:

```
do
{
  declaração;
} while (condição);
```

Mesmo que a declaração seja apenas um comando é uma boa prática deixar as chaves. O ponto-e-vírgula final é obrigatório. Vamos, como anteriormente, ver o funcionamento da estrutura **do-while**:

```
declaração;
if (condição) "Volta para a declaração"
```

Vemos pela análise do bloco acima que a estrutura **do-while** executa a declaração, testa a condição e, se esta for verdadeira, volta para a declaração. A grande novidade no comando **do-while** é que ele, ao contrário do **for** e do **while**, garante que a declaração será executada pelo menos uma vez. Um dos usos da estrutura **do-while** é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido, conforme apresentado abaixo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int i;
      do
      {
            printf ("\n\nEscolha a fruta pelo numero:\n\n");
            printf ("\t(1)...Mamao\n");
            printf ("\t(2)...Abacaxi\n");
            printf ("\t(3)...Laranja\n\n");
            scanf("%d", &i);
      }
      while ((i<1)||(i>3));
      switch (i)
      {
            case 1: printf ("\t\tVoce escolheu Mamao.\n");
            break;
            case 2: printf ("\t\tVoce escolheu Abacaxi.\n");
            break;
            case 3: printf ("\t\tVoce escolheu Laranja.\n");
            break;
      }
      system("pause");
      return(0);
}
```


3.6 *O Comando break*

Nós já vimos dois usos para o comando **break:** interrompendo os comandos **switch** e **for**. Na verdade, estes são os dois usos do comando **break**: ele pode quebrar a execução de um comando (como no caso do **switch**) ou interromper a execução de *qualquer laço* (**for**, **while** ou **do while**). O **break** faz com que a execução do programa continue na primeira linha seguinte ao laço ou bloco que está sendo interrompido.

Observe que um **break** causará uma saída *somente do laço mais interno*. Por exemplo:

```
for(t=0; t<100; ++t)
{
      cont=1;
      for(;;)
      {
            printf("%d", cont);
            cont = cont + 1;
            if(cont==10) break;
      }
}
```

O código acima imprimirá os números de 1 a 10 cem vezes na tela. Toda vez que o **break** é encontrado, o controle é devolvido para o laço **for** externo.

Outra observação é o fato que um **break** usado dentro de uma declaração **switch** afetará somente os dados relacionados com o **switch** e não qualquer outro laço em que o **switch** estiver.

3.7 *O Comando continue*

O comando **continue** pode ser visto como sendo o oposto do **break**. Ele só funciona dentro de um laço. Quando o comando **continue** é encontrado, o laço pula para a próxima iteração, sem o abandono do laço. O programa abaixo exemplifica o uso do *continue*:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int opcao;
      while (opcao != 5)
      {
        printf("\n\n Escolha uma opcao entre 1 e 5: ");
        scanf("%d", &opcao);
        if ((opcao > 5)||(opcao <1)) continue;
                    /* Opcao invalida: volta ao inicio do loop */
        switch (opcao)
        {
            case 1: printf("\n --> Primeira opcao..");
            break;
            case 2: printf("\n --> Segunda opcao..");
            break;
            case 3: printf("\n --> Terceira opcao..");
            break;
            case 4: printf("\n --> Quarta opcao..");
            break;
            case 5: printf("\n --> Abandonando..");
            break;
        }
      }
      system("pause");
      return(0);
}
```

O programa acima ilustra uma aplicação simples para o **continue**. Ele recebe uma opção do usuário. Se esta opção for inválida, o **continue** faz com que o fluxo seja desviado de volta ao início do laço. Caso a opção escolhida seja válida o programa segue normalmente.

3.8 O Comando goto

Vamos mencionar o **goto** apenas para que você saiba que ele existe. Ele pertence a uma classe à parte: a dos comandos de salto incondicional. O **goto** realiza um salto para um local especificado. Este local é determinado por um rótulo. Um rótulo, na linguagem C, é uma marca no programa. Você dá o nome que quiser a esta marca. Podemos tentar escrever uma forma geral:

```
nome_do_rótulo:
  ....
  goto nome_do_rótulo;
  ....
```

Devemos declarar o nome do rótulo, seguido de **:**, na posição para a qual vamos dar o salto. O **goto** pode saltar para um rótulo que esteja para frente ou para trás no programa. Uma observação importante é que o rótulo e o **goto** devem estar dentro da mesma *função*. Como exemplo do uso do **goto** vamos reescrever o equivalente ao comando **for**:

```
inicialização;
inicio:
  if (condição)
  {
        declaração;
        incremento;
        goto inicio;
  }
```

O comando **goto** deve ser utilizado com contenção, pois o abuso no seu uso tende a tornar o código confuso. O **goto** não é um comando *necessário*, podendo sempre ser substituído por outras estruturas de controle. Um caso em que ele pode ser útil é quando temos vários laços e **if**s aninhados e se queira, por algum motivo, sair dos laços e **if**s de uma vez. Neste caso, um **goto** resolve o problema mais elegantemente que vários **break**s, sem contar que os **break**s exigiriam muito mais testes. Ou seja, neste caso o **goto** é mais elegante e mais rápido. O exemplo anterior pode ser reescrito usando-se o **goto**:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int opcao;
      while (opcao != 5)
      {
        REFAZ:
        printf("\n\n Escolha uma opcao entre 1 e 5: ");
        scanf("%d", &opcao);
        if ((opcao > 5)||(opcao <1)) goto REFAZ;
                  /* Opcao invalida: volta ao rotulo REFAZ */
        switch (opcao)
        {
            case 1: printf("\n --> Primeira opcao..");
            break;
            case 2: printf("\n --> Segunda opcao..");
            break;
            case 3: printf("\n --> Terceira opcao..");
            break;
            case 4: printf("\n --> Quarta opcao..");
            break;
            case 5: printf("\n --> Abandonando..");
            break;
        }
      }
      system("pause");
      return(0);
}
```


3.8.1 Exercício

Escreva um programa que peça três inteiros, correspondentes a dia, mês e ano. Peça os números até conseguir valores que estejam na faixa correta (dia entre 1 e 31, mês entre 1 e 12 e ano entre 1900 e 2100). Verifique se o mês e o número de dias estão coerentes (incluindo verificação de anos bissextos). Se estiver tudo certo, imprima o número que aquele dia corresponde no ano. Comente seu programa. PS: Um ano é bissexto se for divisível por 4 e não for divisível por 100, exceto para os anos divisíveis por 400, que também são bissextos.

4 VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES

Há três lugares nos quais podemos declarar variáveis. O primeiro é fora de todas as funções do programa. Estas variáveis são chamadas **variáveis globais** e podem ser usadas a partir de qualquer lugar no programa. Pode-se dizer que, como elas estão fora de todas as funções, todas as funções as vêem. O segundo lugar no qual se pode declarar variáveis é no início de um bloco de código. Estas variáveis são chamadas **locais** e só têm validade dentro do bloco no qual são declaradas, isto é, só a função à qual ela pertence sabe da existência desta variável, dentro do bloco no qual foram declaradas. O terceiro lugar onde se pode declarar variáveis é na lista de parâmetros de uma função. Mais uma vez, apesar de estas variáveis receberem valores externos, estas variáveis são conhecidas apenas pela função onde são declaradas.

Veja o programa abaixo:

```
#include <stdio.h>
#include <stdlib.h>
int contador;
int func1(int j)
{
  ...
}
int main()
{
      char condicao;
      int i;
      for (i=0; ...)
      { /* Bloco do for */
        float f2;
        ...
        func1(i);
      }
      ...
      return(0);
  }
```

A variável *contador* e uma variável global, e é acessível de qualquer parte do programa. As variáveis *condição* e *i*, só existem dentro de **main()**, isto é são variáveis locais de **main**. A variável **float** *f2* é um exemplo de uma variável de bloco, isto é, ela somente é conhecida dentro do bloco do **for**, pertencente à função **main**. A variável inteira *j* é um exemplo de declaração na lista de parâmetros de uma função (a função *func1*).

As regras que regem *onde* uma variável é válida chamam-se regras de *escopo* da variável. Há mais dois detalhes que devem ser ressaltados. Duas variáveis globais não podem ter o mesmo nome. O mesmo vale para duas variáveis locais de uma mesma função. Já duas variáveis locais, de funções diferentes, podem ter o mesmo nome sem perigo algum de conflito.

4.1 Constantes

Constantes são valores que são mantidos fixos pelo compilador. Já usamos constantes neste curso. São consideradas constantes, por exemplo, os números e caracteres como 45.65 ou 'n', etc...

4.1.1 Constantes dos tipos básicos

Abaixo vemos as constantes relativas aos tipos básicos do C:

Tipo de Dado	Exemplos de Constantes
char	'b' '\n' '\0'
int	2 32000 -130
long int	100000 -467

short int	100 -30
unsigned int	50000 35678
float	0.0 23.7 -12.3e-10
double	12546354334.0 -0.0000034236556

4.1.2 Constantes hexadecimais e octais

Muitas vezes precisamos inserir constantes hexadecimais (base dezesseis) ou octais (base oito) em um programa. O C permite que se faça isto. As constantes hexadecimais começam com **0x**. As constantes octais começam em 0. Alguns exemplos:

Constante	Tipo
0xEF	Constante Hexadecimal (8 bits)
0x12A4	Constante Hexadecimal (16 bits)
03212	Constante Octal (12 bits)
034215432	Constante Octal (24 bits)

Portanto nunca escreva 013 achando que o C vai compilar isto como se fosse 13. Na linguagem C *013 é diferente de 13*!

4.1.3 Constantes strings

Já mostramos como o C trata strings. Vamos agora alertar para o fato de que uma string **"Joao"** é na realidade uma constante string. Isto implica, por exemplo, no fato de que **'t'** é diferente de **"t"**, pois **'t'** é um **char** enquanto que **"t"** é uma constante string com dois **chars** onde o primeiro é **'t'** e o segundo é **'\0'**.

4.1.4 Operadores Aritméticos e de Atribuição

Os operadores aritméticos são usados para desenvolver operações matemáticas. A seguir apresentamos a lista dos operadores aritméticos do C:

Operador	Ação
+	Soma (inteira e ponto flutuante)
-	Subtração ou Troca de sinal (inteira e ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Resto de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
--	Decremento (inteiro e ponto flutuante)

O C possui operadores unários e binários. Os unários agem sobre uma variável apenas, modificando ou não o seu valor, e retornam o valor final da variável. Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais. A soma é um operador binário, pois pega duas variáveis, soma seus valores, sem alterar as variáveis, e retorna esta soma. Outros operadores binários são os operadores - (subtração), *****, / e **%**. O operador - como troca de sinal é um operador unário que não altera a variável sobre a qual é aplicado, pois ele retorna o valor da variável multiplicado por -1.

O operador / (divisão) quando aplicado a variáveis inteiras, nos fornece o resultado da divisão inteira; quando aplicado a variáveis em ponto flutuante nos fornece o resultado da divisão "real". Assim seja o seguinte trecho de código:

```
int a = 17, b = 3;
int x, y;
float z = 17. , z1, z2;
x = a / b;
y = a % b;
z1 = z / b;
z2 = a/b;
```

ao final da execução destas linhas, os valores calculados seriam: x = 5, y = 2, z1 = 5.666666 e z2 = 5.0 . Note que na linha correspondente a z2, primeiramente é feita uma divisão inteira (pois os dois operandos são inteiros). Somente depois de efetuada a divisão é que o resultado é atribuído a uma variável **float**.

Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles fazem é incrementar ou decrementar, a variável sobre a qual estão aplicados, de 1. Então

```
x++;
x--;
```

são equivalentes a

```
x=x+1;
x=x-1;
```

Estes operadores podem ser pré-fixados ou pós- fixados. A diferença é que quando são pré-fixados eles incrementam e retornam o valor da variável já incrementada. Quando são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável. Então, em

```
x=23;
y=x++;
```

teremos, no final, **y=23** e **x=24**. Em

```
x=23;
y=++x;
```

teremos, no final, **y=24** e **x=24**. Uma curiosidade: a linguagem de programação C++ tem este nome pois ela seria um "incremento" da linguagem C padrão. A linguagem C++ é igual a linguagem C só que com extensões que permitem a programação orientada a objeto, o que é um recurso extra.

O operador de atribuição do C é o **=**. O que ele faz é pegar o valor à direita e atribuir à variável da esquerda. Além disto ele retorna o valor que ele atribuiu. Isto faz com que as seguintes expressões sejam válidas:

```
x=y=z=1.5; /* Expressao 1 */
if (k=w) ... /* Expressao 2 */
```

A expressão 1 é válida, pois quando fazemos **z=1.5** ela retorna 1.5, que é passado adiante. A expressão dois será verdadeira se **w** for diferente de zero, pois este será o valor retornado por **k=w**. Pense bem antes de usar a expressão dois, pois ela pode gerar erros de interpretação. Você *não* está comparando **k** e **w**. Você está atribuindo o valor de **w** a **k** e usando este valor para tomar a decisão.

4.1.5 Exercícios

a) Diga o resultado das variáveis x, y e z depois da seguinte seqüência de operações:

```
int x,y,z;
x=y=10;
z=++x;
x=-x;
y++;
x=x+y-(z--);
```

4.2 Operadores Relacionais e Lógicos

Os operadores relacionais do C realizam *comparações* entre variáveis. São eles:

Operador	Ação
>	Maior do que
>=	Maior ou igual a
<	Menor do que
<=	Menor ou igual a
==	Igual a

!=	Diferente de

Os operadores relacionais retornam verdadeiro (1) ou falso (0). Para verificar o funcionamento dos operadores relacionais, execute o programa abaixo:

```
/* Este programa ilustra o funcionamento dos operadores relacionais.
*/
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, j;
    printf("\nEntre com dois números inteiros: ");
    scanf("%d%d", &i, &j);
    printf("\n%d == %d é %d\n", i, j, i==j);
    printf("\n%d != %d é %d\n", i, j, i!=j);
    printf("\n%d <= %d é %d\n", i, j, i<=j);
    printf("\n%d >= %d é %d\n", i, j, i>=j);
    printf("\n%d < %d é %d\n", i, j, i<j);
    printf("\n%d > %d é %d\n", i, j, i>j);
    system("pause");
    return(0);
}
```

Você pode notar que o resultado dos operadores relacionais é sempre igual a 0 (falso) ou 1 (verdadeiro).

Para fazer *operações com valores lógicos* (verdadeiro e falso) temos *os operadores lógicos*:

Operador	Ação
&&	AND (E)
||	OR (OU)
!	NOT (NÃO)

Usando os operadores relacionais e lógicos podemos realizar uma grande gama de testes. A tabela-verdade destes operadores é dada a seguir:

p	q	p AND q	p OR q
falso	falso	falso	falso
falso	verdadeiro	falso	verdadeiro
verdadeiro	falso	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro	verdadeiro

O programa a seguir ilustra o funcionamento dos operadores lógicos. Compile-o e faça testes com vários valores para i e j:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, j;
    printf("informe dois números(cada um sendo 0 ou 1): ");
    scanf("%d%d", &i, &j);
    printf("%d AND %d é %d\n", i, j, i && j);
    printf("%d OR %d é %d\n", i, j, i || j);
    printf("NOT %d é %d\n", i, !i);
    system("pause");
}
```

Exemplo: No trecho de programa abaixo a operação j++ será executada, pois o resultado da expressão lógica é verdadeiro:

```
int i = 5, j =7;
if ( (i > 3) && ( j <= 7) && ( i != j) ) j++;
        V    AND    V     AND    V = V
```

Mais um exemplo. O programa abaixo, imprime na tela somente os números pares entre 1 e 100, apesar da variação de i ocorrer de 1 em 1:

```
/* Imprime os números pares entre 1 e 100. */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i;
    for(i=1; i<=100; i++)
        if(!(i%2)) printf("%d ",i); /* o operador de resto */
        system("pause");      /* dará falso (zero) quando */
                              /* usada com número par. */
}                        /* Esse resultado é invertido pelo ! */
```

4.2.1 Operadores Lógicos Bit a Bit

O C permite que se faça *operações lógicas "bit-a-bit"* em números. Ou seja, neste caso, o número é representado por sua forma binária e as operações são feitas em cada bit dele. Imagine um número inteiro de 16 bits, a variável *i*, armazenando o valor 2. A representação binária de *i*, será: 0000000000000010 (quinze zeros e um único 1 na segunda posição da direita para a esquerda). Poderemos fazer operações em cada um dos bits deste número. Por exemplo, se fizermos a negação do número (operação binária NOT, ou operador binário ~ em C), isto é, ***~i***, o número se transformará em 1111111111111101. As operações binárias ajudam programadores que queiram trabalhar com o computador em "baixo nível". As operações lógicas bit a bit só podem ser usadas nos tipos **char**, **int** e **long int**. Os operadores são:

Operador	Ação
&	AND
|	OR
^	XOR (OR exclusivo)
~	NOT
>>	Deslocamento de bits à direita
<<	Deslocamento de bits à esquerda

Os operadores **&**, |, ^ e **~** são as operações lógicas bit a bit. A forma geral dos operadores de deslocamento é:

```
valor>>número_de_deslocamentos
valor<<número_de_deslocamentos
```

O número_de_deslocamentos indica o quanto cada bit irá ser deslocado. Por exemplo, para a variável i anterior, armazenando o número 2:

```
i << 3;
```

fará com que i agora tenha a representação binária: 0000000000010000, isto é, o valor armazenado em i passa a ser igual a 16.

4.2.2 Exercícios

Diga se as seguintes expressões serão verdadeiras ou falsas:

```
((10>5)||(5>10))
(!(5==6)&&(5!=6)&&((2>1)||(5<=4)))
```

4.3 Expressões

Expressões são combinações de variáveis, constantes e operadores. Quando montamos expressões temos que levar em consideração a ordem com que os operadores são executados.

Exemplos de expressões:

```
Anos=Dias/365.25;
i = i+3;
```



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

4.3.1 Conversão de tipos em expressões

Quando o C avalia expressões onde temos variáveis de tipos diferentes o compilador verifica se as conversões são possíveis. Se não são, ele não compilará o programa, dando uma mensagem de erro. Se as conversões forem possíveis ele as faz, seguindo as regras abaixo:

- ✓ Todos os **char**s e **short int**s são convertidos para **int**s. Todos os **float**s são convertidos para **double**s.
- ✓ Para pares de operandos de tipos diferentes: se um deles é **long double** o outro é convertido para **long double**; se um deles é **double** o outro é convertido para **double**; se um é **long** o outro é convertido para **long**; se um é **unsigned** o outro é convertido para **unsigned**.

4.3.2 - Expressões que Podem ser Abreviadas

O C admite as seguintes equivalências, que podem ser usadas para simplificar expressões ou para facilitar o entendimento de um programa:

Expressão Original	Expressão Equivalente
x=x+k;	x+=k;
x=x-k;	x-=k;
x=x*k;	x*=k;
x=x/k;	x/=k;
x=x>>k;	x>>=k;
x=x<<k;	x<<=k;
x=x&k;	x&=k;
etc...	

4.3.3 Encadeando expressões: o operador ,

O operador **,** determina uma lista de expressões que devem ser executadas seqüencialmente. Em síntese, a vírgula diz ao compilador: execute as duas expressões separadas pela vírgula, em seqüência. O valor retornado por uma expressão com o operador **,** é sempre dado pela expressão mais à direita. No exemplo abaixo:

```
x=(y=2,y+3);
```

o valor 2 vai ser atribuído a **y**, se somará 3 a **y** e o retorno (5) será atribuído à variável **x**. Pode-se encadear quantos operadores **,** forem necessários.

O exemplo a seguir mostra um outro uso para o operador **,** dentro de um **for**:

```
#include<stdio.h>
#include <stdlib.h>
int main()
{
    int x, y;
    for(x=0 , y=0 ;  x+y < 100 ; ++x , y++)
          /* Duas variáveis de controle: x e y . Foi atribuído o valor
          zero a cada uma delas na inicialização do for e ambas são
          incrementadas na parte de incremento do for */
    printf("\n%d ", x+y);
          /* o programa imprimirá os números pares de 2 a 98 */
    system("pause");
}
```

4.3.4 Tabela de Precedências do C

Esta é a tabela de precedência dos operadores em C. Alguns (poucos) operadores ainda não foram estudados, e serão apresentados em aulas posteriores.

Maior precedência	() [] ->
	! ~ ++ -- . -(unário) (cast) *(unário) &(unário) sizeof
	* / %
	+ -
	<< >>
	<<= >>=
	== !=
	&
	^
	|
	&&
	||
	?
	= += -= *= /=
Menor precedência	,

Uma dica aos iniciantes: Você não precisa saber toda a tabela de precedências de cor. É útil que você conheça as principais relações, mas é aconselhável que ao escrever o seu código, você tente isolar as expressões com parênteses, para tornar o seu programa mais legível.

4.4 Modeladores (Casts)

Um modelador é aplicado a uma expressão. Ele *força* a mesma a ser de um tipo especificado. Sua forma geral é:

```
(tipo)expressão
```

Um exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
   int num;
   float f;
   num=10;
   f=(float)num/7; /* Uso do modelador. Força a
                      transformação de num em um float */
   printf ("%f",f);
   system("pause");
   return(0);
}
```

Se não tivéssemos usado o modelador no exemplo acima o C faria uma divisão inteira entre 10 e 7. O resultado seria 1 (um) e este seria depois convertido para **float** mas continuaria a ser 1.0. Com o modelador temos o resultado correto.

4.4.1 Exercícios

Compile o exemplo acima sem usar o modelador, e verifique os resultados. Compile-o novamente usando o modelador e compare a saída com os resultados anteriores.

5 MATRIZES E STRINGS

5.1 Vetores

Vetores nada mais são que matrizes unidimensionais. Vetores são uma estrutura de dados muito utilizada. É importante notar que vetores, matrizes bidimensionais e matrizes de qualquer dimensão são caracterizadas por terem todos os elementos pertencentes ao mesmo tipo de dado. Para se declarar um vetor podemos utilizar a seguinte forma geral:

```
tipo_da_variável nome_da_variável [tamanho];
```


Quando o C vê uma declaração como esta ele reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos:

```
float exemplo [20];
```

o C irá reservar 4x20=80 bytes. Estes bytes são reservados de maneira adjacente. Na linguagem C a numeração começa sempre em zero. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessá-los vamos escrever:

```
exemplo[0]
exemplo[1]
...
exemplo[19]
```

Mas ninguém o impede de escrever:

```
exemplo[30]
exemplo[103]
```

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que *você* deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter variáveis sobrescritas ou de ver o computador travar. *Bugs* terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num[100]; /* Declara um vetor de inteiros de 100 posicoes */
      int count=0;
      int totalnums;
      do
      {
            printf ("\nEntre com um numero (-999 p/ terminar): ");
            scanf ("%d",&num[count]);
            count++;
      }
      while (num[count-1]!=-999);
      totalnums=count-1;
      printf ("\n\n\n\t Os números que você digitou foram:\n\n");
      for (count=0;count<totalnums;count++)
      printf (" %d",num[count]);
      system("pause");
      return(0);
}
```

No exemplo acima, o inteiro *count* é inicializado em 0. O programa pede pela entrada de números até que o usuário entre com o Flag -999. Os números são armazenados no vetor *num*. A cada número armazenado, o contador do vetor é incrementado para na próxima iteração escrever na próxima posição do vetor. Quando o usuário digita o flag, o programa abandona o primeiro *loop* e armazena o total de números gravados. Por fim, todos os números são impressos. É bom lembrar aqui que nenhuma restrição é feita quanto à quantidade de números digitados. Se o usuário digitar mais de 100 números, o programa tentará ler normalmente, mas o programa os escreverá em uma parte não alocada de memória, pois o espaço alocado foi para somente 100 inteiros. Isto pode resultar nos mais variados erros no instante da execução do programa.

5.1.1 Exercícios

Reescreva o exemplo acima, realizando a cada leitura um teste para ver se a dimensão do vetor não foi ultrapassada. Caso o usuário entre com 100 números, o programa deverá abortar o *loop* de leitura automaticamente. O uso do Flag (-999) não deve ser retirado.

5.2 *Strings*

Strings são vetores de **char**s. Nada mais e nada menos. As strings são o uso mais comum para os vetores. Devemos apenas ficar atentos para o fato de que as strings têm o seu último elemento como um **\0**. A declaração geral para uma string é:

```
char nome_da_string [tamanho];
```

Devemos lembrar que o tamanho da string deve incluir o **\0** final. A biblioteca padrão do C possui diversas funções que manipulam strings. Estas funções são úteis, pois não se pode, por exemplo, igualar duas strings:

```
string1=string2; /* NAO faca isto */
```

Fazer isto é um desastre. Quando você terminar de ler a seção que trata de ponteiros você entenderá porquê. As strings devem ser igualadas elemento a elemento.

Quando vamos fazer programas que tratam de string muitas vezes podemos fazer bom proveito do fato de que uma string termina com **\0** (isto é, o número inteiro 0). Veja, por exemplo, o programa abaixo que serve para igualar duas strings (isto é, copia os caracteres de uma string para o vetor da outra):

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int count;
      char str1[100],str2[100];
      ..../* Aqui o programa le str1 que sera copiada para str2 */
      for (count=0;str1[count];count++)
      str2[count]=str1[count];
      str2[count]='\0';
      .... /* Aqui o programa continua */
}
```

A condição no *loop* **for** acima é baseada no fato de que a string que está sendo copiada termina com **\0**. Quando o elemento encontrado em **str1[count]** é o **\0**, o valor retornado para o teste condicional é falso (nulo). Desta forma a expressão que vinha sendo verdadeira (não zero) continuamente, torna-se falsa.

Vamos ver agora algumas funções básicas para manipulação de strings.

5.2.1 *gets*

A função **gets()** lê uma string do teclado. Sua forma geral é:

```
gets (nome_da_string);
```

O programa abaixo demonstra o funcionamento da função **gets()**:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      char string[100];
      printf ("Digite o seu nome: ");
      gets (string);
      printf ("\n\n Ola %s",string);
      system("pause");
      return(0);
}
```

Repare que é válido passar para a função **printf()** o nome da string. Você verá mais adiante porque isto é válido. Como o primeiro argumento da função **printf()** é uma string também é válido fazer:


```
printf (string);
```

isto simplesmente imprimirá a string.

5.2.2 *strcpy*

Sua forma geral é:

```
strcpy (string_destino,string_origem);
```

A função **strcpy()** copia a string-origem para a string-destino. Seu funcionamento é semelhante ao da rotina apresentada na seção anterior. As funções apresentadas nestas seções estão no arquivo cabeçalho **string.h**. A seguir apresentamos um exemplo de uso da função **strcpy()**:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main ()
{
      char str1[100],str2[100],str3[100];
      printf ("Entre com uma string: ");
      gets (str1);
      strcpy (str2,str1);     /* Copia str1 em str2 */
      strcpy (str3,"Voce digitou a string "); /* Copia "Voce
                                        digitou a string" em str3 */
      printf ("\n\n%s%s",str3,str2);
      system("pause");
      return(0);
}
```

5.2.3 *strcat*

A função **strcat()** tem a seguinte forma geral:

```
strcat (string_destino,string_origem);
```

A string de origem permanecerá inalterada e será anexada ao fim da string de destino. Um exemplo:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main ()
{
      char str1[100],str2[100];
      printf ("Entre com uma string: ");
      gets (str1);
      strcpy (str2,"Voce digitou a string ");
      strcat (str2,str1);     /* str2 armazenara' Voce digitou a
                                 string + o conteudo de str1 */
      printf ("\n\n%s",str2);
      system("pause");
      return(0);
}
```

5.2.4 *strlen*

Sua forma geral é:

```
strlen (string);
```

A função **strlen()** retorna o comprimento da string fornecida. O terminador nulo não é contado. Isto quer dizer que, de fato, o comprimento do vetor da string deve ser um a mais que o inteiro retornado por **strlen()**. Um exemplo do seu uso:

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

```
int main ()
{
      int size;
      char str[100];
      printf ("Entre com uma string: ");
      gets (str);
      size=strlen (str);
      printf ("\n\nA string que voce digitou tem tamanho %d",size);
      return(0);
}
```

5.2.5 *strcmp*

Sua forma geral é:

```
strcmp (string1,string2);
```

A função **strcmp()** compara a string 1 com a string 2. Se as duas forem idênticas a função retorna zero. Se elas forem diferentes a função retorna não-zero. Um exemplo da sua utilização:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main ()
{
      char str1[100],str2[100];
      printf ("Entre com uma string: ");
      gets (str1);
      printf ("\n\nEntre com outra string: ");
      gets (str2);
      if (strcmp(str1,str2))
      printf ("\n\nAs duas strings são diferentes.");
      else printf ("\n\nAs duas strings são iguais.");
      system("pause");
      return(0);
}
```

5.2.6 *Exercícios*

Faça um programa que leia quatro palavras pelo teclado, e armazene cada palavra em uma string. Depois, concatene todas as strings lidas numa única string. Por fim apresente esta como resultado ao final do programa.

5.3 *Matrizes*

5.3.1 *Matrizes bidimensionais*

Já vimos como declarar matrizes unidimensionais (vetores). Vamos tratar agora de matrizes bidimensionais. A forma geral da declaração de uma matriz bidimensional é muito parecida com a declaração de um vetor:

```
tipo_da_variável nome_da_variável [altura][largura];
```

É muito importante ressaltar que, nesta estrutura, o índice da esquerda indexa as linhas e o da direita indexa as colunas. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia mais rapidamente que o índice à esquerda. Mais uma vez é bom lembrar que, na linguagem C, os índices variam de zero ao valor declarado, menos um; mas o C não vai verificar isto para o usuário. Manter os índices na faixa permitida é tarefa do programador. Abaixo damos um exemplo do uso de uma matriz:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
int mtrx [20][10];
```



```
int i,j,count;
count=1;
for (i=0;i<20;i++)
      for (j=0;j<10;j++)
      {
            mtrx[i][j]=count;
            count++;
      }
      system("pause");
      return(0);
}
```

No exemplo acima, a matriz **mtrx** é preenchida, sequencialmente por linhas, com os números de 1 a 200. Você deve entender o funcionamento do programa acima antes de prosseguir.

5.3.2 *Matrizes de strings*

Matrizes de strings são matrizes bidimensionais. Imagine uma string. Ela é um vetor. Se fizermos um vetor de strings estaremos fazendo uma lista de vetores. Esta estrutura é uma matriz bidimensional de **char**s. Podemos ver a forma geral de uma matriz de strings como sendo:

```
char nome_da_variável [num_de_strings][compr_das_strings];
```

Aí surge a pergunta: como acessar uma string individual? Fácil. É só usar apenas o primeiro índice. Então, para acessar uma determinada string faça:

```
nome_da_variável [índice]
```

Aqui está um exemplo de um programa que lê 5 *strings* e as exibe na tela:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      char strings [5][100];
      int count;
      for (count=0;count<5;count++)
      {
            printf ("\n\nDigite uma string: ");
            gets (strings[count]);
      }
      printf ("\n\n\nAs strings que voce digitou foram:\n\n");
      for (count=0;count<5;count++)
            printf ("%s\n",strings[count]);
      system("pause");
      return(0);
}
```

5.3.3 *Matrizes multidimensionais*

O uso de matrizes multidimensionais na linguagem C é simples. Sua forma geral é:

```
tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];
```

Uma matriz N-dimensional funciona basicamente como outros tipos de matrizes. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

5.3.4 *Inicialização*

Podemos inicializar matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz com inicialização é:

```
tipo_da_variável   nome_da_variável   [tam1][tam2]   ...   [tamN]   =
{lista_de_valores};
```

A lista de valores é composta por valores (do mesmo tipo da variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz. Abaixo vemos alguns exemplos de inicializações de matrizes:

```
float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matrx [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
char str [10] = "Joao";
char str_vect [3][10] = { "Joao", "Maria", "Jose" };
```

O primeiro demonstra inicialização de vetores. O segundo exemplo demonstra a inicialização de matrizes multidimensionais, onde **matrx** está sendo inicializada com 1, 2, 3 e 4 em sua primeira linha, 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na última linha. No terceiro exemplo vemos como inicializar uma string e, no quarto exemplo, um modo mais compacto de inicializar uma string. O quinto exemplo combina as duas técnicas para inicializar um vetor de strings. Repare que devemos incluir o **;** no final da inicialização.

5.3.5 Inicialização sem especificação de tamanho

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho *a priori*. O compilador C vai, neste caso verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa, sendo muito útil, por exemplo, quando vamos inicializar uma string e não queremos contar quantos caracteres serão necessários. Alguns exemplos:

```
char mess [] = "Linguagem C: flexibilidade e poder.";
int matrx [][2] = { 1,2,2,4,3,6,4,8,5,10 };
```

No primeiro exemplo, a string mess terá tamanho 36. Repare que o artifício para realizar a inicialização sem especificação de tamanho é não especificar o tamanho! No segundo exemplo o valor não especificado será 5.

5.3.6 Exercícios

a) O que imprime o programa a seguir? Tente entendê-lo e responder. A seguir, execute-o e comprove o resultado.

```
# include <stdio.h>
#include <stdlib.h>
int main()
{
      int t, i, M[3][4];
      for (t=0; t<3; ++t)
            for (i=0; i<4; ++i)
                  M[t][i] = (t*4)+i+1;
      for (t=0; t<3; ++t)
      {
            for (i=0; i<4; ++i)
                  printf ("%3d ", M[t][i]);
            printf ("\n");
      }
      system("pause");
      return(0);
}
```

6 PONTEIROS

O C é altamente dependente dos ponteiros. Para ser um bom programador em C é fundamental que se tenha um bom domínio deles. Por isto, é recomendável o aluno ter um carinho especial com esta parte do curso que trata deles. Ponteiros são tão importantes na linguagem C que você já os viu e nem percebeu, pois mesmo para se fazer uma introdução básica à linguagem C precisa-se deles.

Mas, lembre-se: *o uso descuidado de ponteiros pode levar a sérios bugs e a dores de cabeça terríveis.*

6.1 Como Funcionam os Ponteiros

Os **int**s guardam inteiros. Os **float**s guardam números de ponto flutuante. Os **char**s guardam caracteres. Ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.

Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de *tipos* diferentes.

No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro **int** aponta para um inteiro, isto é, guarda o endereço de um inteiro.

6.2 Declarando e Utilizando Ponteiros

Para declarar um ponteiro temos a seguinte forma geral:

```
tipo_do_ponteiro *nome_da_variável;
```

É o asterisco (*****) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;
char *temp,*pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior. *O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado!* Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count=10;
int *pt;
pt=&count;
```

Criamos um inteiro **count** com o valor 10 e um apontador para um inteiro **pt**. A expressão **&count** nos dá o endereço de **count**, o qual armazenamos em **pt**. Simples, não é? Repare que *não* alteramos o valor de **count**, que continua valendo 10.

Como nós colocamos um endereço em **pt**, ele está agora *liberado* para ser usado. Podemos, por exemplo, alterar o valor de **count** usando **pt**. Para tanto vamos usar o operador *inverso* do

operador **&**, que é o operador *****. No exemplo acima, uma vez que fizemos **pt=&count** a expressão ***pt** é equivalente ao próprio **count**. Isto significa que, se quisermos mudar o valor de **count** para 12, basta fazer ***pt=12**.

Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo. Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador **&**. Ou seja, o operador **&** aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador ***** ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

Uma observação importante: apesar do símbolo ser o mesmo, o operador ***** (multiplicação) não é o mesmo operador que o ***** (referência de ponteiros). Para começar, o primeiro é binário, e o segundo é unário pré-fixado.

Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num,valor;
      int *p;
      num=55;
      p=&num; /* Pega o endereco de num */
      valor=*p;// Valor e igualado a num de uma maneira indireta
      printf ("\n\n%d\n",valor);
      printf ("Endereco para onde o ponteiro aponta: %p\n",p);
      printf ("Valor da variavel apontada: %d\n",*p);
      system("pause");
      return(0);
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int num,*p;
      num=55;
      p=&num; /* Pega o endereco de num */
      printf ("\nValor inicial: %d\n",num);
      *p=100; /* Muda o valor de num de uma maneira indireta */
      printf ("\nValor final: %d\n",num);
      system("pause");
      return(0);
}
```

Nos exemplos acima vemos um primeiro exemplo do funcionamento dos ponteiros. No primeiro exemplo, o código **%p** usado na função **printf()** indica à função que ela deve imprimir um endereço.

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se tivermos dois ponteiros **p1** e **p2** podemos igualá-los fazendo **p1=p2**. Repare que estamos fazendo com que **p1** aponte para o mesmo lugar que **p2**. Se quisermos que a variável apontada por **p1** tenha o mesmo conteúdo da variável apontada por **p2** devemos fazer ***p1=*p2**. Basicamente, depois que se aprende a usar os dois operadores (**&** e *****) fica fácil entender operações com ponteiros.

As próximas operações, também muito usadas, são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o

ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro **char*** ele anda 1 byte na memória e se você incrementa um ponteiro **double*** ele anda 8 bytes na memória. O decremento funciona semelhantemente. Supondo que **p** é um ponteiro, as operações são escritas como:

```
p++;
p--;
```

Mais uma vez insiste-se. Estamos falando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **p**, faz-se:

```
(*p)++;
```

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer:

```
p=p+15; ou p+=15;
```

E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);
```

A subtração funciona da mesma maneira. Uma outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros? Bem, em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes (== e **!**=). No caso de operações do tipo >, <, >= e <= estamos comparando qual ponteiro aponta para uma posição mais alta *na memória*. Então uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
p1>p2
```

Entretanto há operações que você *não* pode efetuar num ponteiro. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair **float**s ou **double**s de ponteiros.

6.2.1 Exercícios

a) Explique a diferença entre

```
p++; (*p)++; *(p++);
```

O que quer dizer *(p+10);?

Explique o que você entendeu da comparação entre ponteiros

b) Qual o valor de y no final do programa? Tente primeiro descobrir e depois verifique no computador o resultado. A seguir, escreva um /* comentário */ em cada comando de atribuição explicando o que ele faz e o valor da variável à esquerda do '=' após sua execução.

```
#include <stdlib.h>
int main()
{
      int y, *p, x;
      y = 0;
      p = &y;
      x = *p;
      x = 4;
      (*p)++;
      x--;
      (*p) += x;
      printf ("y = %d\n", y);
      system("pause");
      return(0);
}
```

6.3 *Ponteiros e Vetores*

Veremos nestas seções que ponteiros e vetores têm uma ligação muito forte.

6.3.1 *Vetores como ponteiros*

Vamos dar agora uma idéia de como o C trata vetores. Quando você declara uma matriz da seguinte forma:

```
tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];
```

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:

```
tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo
```

O compilador então aloca este número de bytes em um espaço livre de memória. O *nome da variável* que você declarou é na verdade *um ponteiro para o tipo da variável da matriz*. Este conceito é fundamental. Eis porque: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento da matriz. Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?

```
nome_da_variável[índice]
```

Isto pode ser facilmente explicado desde que você entenda que a notação acima é *absolutamente equivalente* a se fazer:

```
*(nome_da_variável+índice)
```

Agora podemos entender como é que funciona um vetor! Vamos ver o que podemos tirar de informação deste fato. Fica claro, por exemplo, porque é que, no C, a indexação começa com zero. É porque, ao pegarmos o valor do primeiro elemento de um vetor, queremos, de fato, ***nome_da_variável** e então devemos ter um índice igual a zero. Então sabemos que:

```
*nome_da_variável é equivalente a nome_da_variável[0]
```

Outra coisa: apesar de, na maioria dos casos, não fazer muito sentido, poderíamos ter índices negativos. Estaríamos pegando posições de memória antes do vetor. Isto explica também porque o C não verifica a validade dos índices. Ele *não* sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

Vamos ver agora um dos usos mais importantes dos ponteiros: a varredura seqüencial de uma matriz. Quando temos que varrer todos os elementos de uma matriz de uma forma seqüencial, podemos usar um ponteiro, o qual vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

```
#include <stdlib.h>
int main ()
{
      float matrx [50][50];
      int i,j;
      for (i=0;i<50;i++)
      for (j=0;j<50;j++)
      matrx[i][j]=0.0;
      system("pause");
      return(0);
}
```

Podemos reescrevê-lo usando ponteiros:

```
#include <stdlib.h>
int main ()
{
      float matrx [50][50];
      float *p;
      int count;
      p=matrx[0];
      for (count=0;count<2500;count++)
      {
            *p=0.0;
            p++;
      }
      system("pause");
      return(0);
}
```


No primeiro programa, *cada* vez que se faz **matrx[i][j]** o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.

Há uma diferença entre o nome de um vetor e um ponteiro que deve ser frisada: um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa, que não se consegue alterar o endereço que é apontado pelo "nome do vetor". Seja:

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;
/* as operacoes a seguir sao invalidas */
vetor = vetor + 2; /* ERRADO: vetor nao e' variavel */
vetor++;           /* ERRADO: vetor nao e' variavel */
vetor = ponteiro;  /* ERRADO: vetor nao e' variavel */
```

Teste as operações acima no seu compilador. Ele dará uma mensagem de erro. Alguns compiladores dirão que vetor não é um Lvalue. Lvalue, significa "Left value", um símbolo que pode ser colocado do lado esquerdo de uma expressão de atribuição, isto é, uma variável. Outros compiladores dirão que se tem "incompatible types in assignment", tipos incompatíveis em uma atribuição.

```
/* as operacoes abaixo sao validas */
ponteiro = vetor;       /* CERTO: ponteiro e' variavel */
ponteiro = vetor+2;     /* CERTO: ponteiro e' variavel */
```

O que você aprendeu nesta seção é de suma importância. Não siga adiante antes de entendê-la bem.

6.3.2 *Ponteiros como vetores*

Sabemos agora que, na verdade, o nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor. Como conseqüência podemos também indexar um ponteiro qualquer. O programa mostrado a seguir funciona perfeitamente:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
      int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      int *p;
      p=matrx;
      printf ("O terceiro elemento do vetor e: %d",p[2]);
      system("pause");
      return(0);
}
```

Podemos ver que **p[2]** equivale a ***(p+2)**.

6.3.3 *Strings*

Seguindo o raciocínio acima, nomes de strings, são do tipo **char***. Isto nos permite escrever a nossa função **StrCpy()**, que funcionará de forma semelhante à função **strcpy()** da biblioteca:

```
#include <stdio.h>
#include <stdlib.h>
void StrCpy (char *destino,char *origem)
{
      while (*origem)
      {
            *destino=*origem;
            origem++;
            destino++;
      }
      *destino='\0';
}
int main ()
{
      char str1[100],str2[100],str3[100];
      printf ("Entre com uma string: ");
      gets (str1);
      StrCpy (str2,str1);
      StrCpy (str3,"Voce digitou a string ");
      printf ("\n\n%s%s",str3,str2);
      system("pause");
      return(0);
}
```

Há vários pontos a destacar no programa acima. Observe que podemos passar ponteiros como argumentos de funções. Na verdade é assim que funções como **gets()** e **strcpy()** funcionam. Passando o ponteiro você possibilita à função *alterar* o conteúdo das strings. Você já estava passando os ponteiros e não sabia. No comando **while (*origem)** estamos usando o fato de que a string termina com **\0** como critério de parada. Quando fazemos **origem**++ e **destino**++ o aluno poderia argumentar que estamos alterando o valor do ponteiro-base da string, contradizendo o que recomendei que se deveria fazer, no final de uma seção anterior. O que o aluno talvez não saiba ainda (e que será estudado em detalhe mais adiante) é que, no C, são passados para as funções *cópias* dos argumentos. Desta maneira, quando alteramos o ponteiro **origem** na função **StrCpy()** o ponteiro **str2** permanece inalterado na função **main()**.

6.3.4 *Endereços de elementos de vetores*

Nesta seção vamos apenas ressaltar que a notação

```
&nome_da_variável[índice]
```

é válida e retorna o endereço do ponto do vetor indexado por índice. Isto seria equivalente a nome_da_variável + índice. É interessante notar que, como conseqüência, o ponteiro **nome_da_variável** tem o endereço **&nome_da_variável[0]**, que indica onde na memória está guardado o valor do primeiro elemento do vetor.

6.3.5 *Vetores de ponteiros*

Podemos construir vetores de ponteiros como declaramos vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser:

```
int *pmatrx [10];
```

No caso acima, **pmatrx** é um vetor que armazena 10 ponteiros para inteiros.

6.3.6 *Exercicios*

Fizemos a função StrCpy(). Faça uma função StrLen() e StrCat() que funcionem como as funções strlen() e strcat() de string.h respectivamente

6.4 *Inicializando Ponteiros*

Podemos inicializar ponteiros. Vamos ver um caso interessante dessa inicialização de ponteiros com strings.

Precisamos, para isto, entender como o C trata as strings constantes. Toda string que o programador insere no programa é colocada num banco de strings que o compilador cria. No local onde está uma string no programa, o compilador coloca o endereço do início daquela string (que está no banco de strings). É por isto que podemos usar **strcpy()** do seguinte modo:

```
strcpy (string,"String constante.");
```

strcpy() pede dois parâmetros do tipo **char***. Como o compilador substitui a string "String constante." pelo seu endereço no banco de strings, tudo está bem para a função **strcpy()**. O que isto tem a ver com a inicialização de ponteiros? É que, para uma string que vamos usar várias vezes, podemos fazer:

```
char *str1="String constante.";
```

Aí poderíamos, em todo lugar que precisarmos da string, usar a variável **str1**. Devemos apenas tomar cuidado ao usar este ponteiro. Se o alterarmos, vamos perder a string. Se o usarmos para alterar a string, podemos facilmente corromper o banco de strings que o compilador criou. Mais uma vez fica o aviso: ponteiros são poderosos, mas se usados com descuido, podem ser uma ótima fonte de dores de cabeça.

6.4.1 *Exercicios*

Escreva a função

```
int strend(char *s, char *t)
```

que retorna 1 (um) se a cadeia de caracteres **t** ocorrer no final da cadeia **s**, e 0 (zero) caso contrário.

6.5 *Ponteiros para Ponteiros*

Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo. Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

```
tipo_da_variável **nome_da_variável;
```

Algumas considerações: ****nome_da_variável** é o conteúdo final da variável apontada; ***nome_da_variável** é o conteúdo do ponteiro intermediário.

No C podemos declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros (UFA!) e assim por diante. Para fazer isto (não me pergunte qual a utilidade disto!) basta aumentar o número de asteriscos na declaracão. A lógica é a mesma.

Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes, como mostrado no exemplo abaixo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      float fpi = 3.1415, *pf, **ppf;
      pf = &fpi;  /* pf armazena o endereco de fpi */
      ppf = &pf;  /* ppf armazena o endereco de pf */
      printf("%f", **ppf); /* Imprime o valor de fpi */
      printf("%f", *pf);   /* Tambem imprime o valor de fpi */
      system("pause");
      return(0);
}
```

6.5.1 *Exercícios*

Verifique o programa abaixo. Encontre o seu erro e corrija-o para que escreva o número 10 na tela.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int x, *p, **q;
      p = &x;
      q = &p;
      x = 10;
      printf("\n%d\n", &q);
      system("pause");
      return(0);
}
```

6.6 *Cuidados a Serem Tomados ao se Usar Ponteiros*

O principal cuidado ao se usar um ponteiro deve ser: saiba sempre *para onde* o ponteiro está apontando. Isto inclui: nunca use um ponteiro que não foi inicializado. Um pequeno programa que demonstra como ***não*** usar um ponteiro:

```
int main () /* Errado - Nao Execute */
{
      int x,*p;
      x=13;
      *p=x;
      return(0);
}
```

Este programa compilará e rodará. O que acontecerá? Ninguém sabe. O ponteiro p pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).

6.6.1 *Exercícios*

Escreva um programa que declare uma matriz 100x100 de inteiros. Você deve inicializar a matriz com zeros usando ponteiros para endereçar seus elementos. Preencha depois a matriz com os números de 1 a 10000, também usando ponteiros.