Material desenvolvido por
Prof. M.Sc. Adriano Brum Fontoura

Linguagem C

I n d i c e

Esta material de apoio foi basedo no conteúdo do site do Curso de Linguagem C da MG (site *http://www.ead.eee.ufmg.br/cursos/C/*). Sendo que esta versão .doc foi elaborada por Henrique José dos Santos (UNISANTOS, Santos-SP), e resumida pelo Prof. Adriano Brum Fontoura,(URI, Santiago-RS), visado atender ao conteúdo programático da disciplina de Linguagem de Programação I no curso de Ciência da Computação.

Este curso foi implementado na UFMG - Universidade Federal de Minas Gerais pelo Núcleo de Ensino à Distância da Escola de Engenharia - fazendo parte de um projeto apoiado pela Pró-Reitoria de Graduação da UFMG, através do programa PROGRAD97/FUNDO-FUNDEP. O curso é oferecido regularmente, a cada semestre, desde 1997. O curso tem sido oferecido gratuitamente e não oferece certificados de conclusão. Ou seja, você deve fazer este curso se estiver interessado em seu aprimoramento pessoal.

Quem originalmente escreveu o curso de C foi o aluno de graduação em Engenharia Elétrica, Daniel Balparda de Carvalho. Algumas modificações foram introduzidas pela aluna de doutorado Ana Liddy Cenni de Castro Magalhães e pelo aluno de graduação em Engenharia Elétrica, Ebenezer Silva Oliveira. Posteriormente, Guilherme Neves Cavalieri, também aluno de graduação em Engenharia Elétrica, modificou as páginas, de forma a facilitar a navegação e utilização do curso. Atualmente ele é mantido pelo professor Renato Cardoso Mesquita.

Aula 1 - INTRODUÇÃO

Vamos, neste curso, aprender os conceitos básicos da linguagem de programação C a qual 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".

Apesar de ser bom, não é pré-requisito do curso um conhecimento anterior de linguagens de programação. É importante uma familiaridade com computadores. O que é importante é que você tenha vontade de aprender, dedicação ao curso e, caso esteja em uma das turmas do curso, acompanhe atentamente as discussões que ocorrem na lista de discussões do curso.

O C nasceu na década de 70. Seu inventor, Dennis Ritchie, implementou-o pela primeira vez usando um DEC PDP-11 rodando o sistema operacional UNIX. O C é derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards.

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 ... É bem provável que o Navegador que você está usando para ler este texto tenha sido escrito em C ou C++.

Estudaremos a estrutura do ANSI C, o C padronizado pela ANSI. Veremos ainda algumas funções comuns em compiladores para alguns sistemas operacionais. Quando não houver equivalentes para as funções em outros sistemas, apresentaremos formas alternativas de uso dos comandos.

Sugerimos que o aluno realmente use o máximo possível dos exemplos, problemas e exercícios aqui 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 leitor digite, compile e execute os exemplos apresentados.

Aula 2 - Primeiros Passos

O C é "Case Sensitive"

Vamos começar o nosso curso ressaltando um ponto de suma importância: o C é "Case Sensitive", isto é, *maiúsculas e minúsculas fazem diferença*. Se declarar uma variável com o nome soma ela será diferente de **Soma**, **SOMA**, **SoMa** ou **sOmA**. Da mesma maneira, os comandos do C **if** e **for**, por exemplo, só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis.

Dois Primeiros Programas

Vejamos um primeiro programa em C:

```
#include <stdio.h>
/* Um Primeiro Programa */
int main ()
{
```

```
    printf ("Ola! Eu estou vivo!\n");
    return(0);
}
```

Compilando e executando este programa você verá que ele coloca a mensagem *Ola! Eu estou vivo!* na tela.

Vamos analisar o programa por partes.

A linha **#include <stdio.h>** diz ao compilador que ele deve incluir o arquivo-cabeçalho **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 diversos Arquivos-cabeçalho.

Quando fazemos um programa, uma boa idéia é 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 */. Um comentário pode, inclusive, ter mais de uma linha.

A linha **int main()** indica que estamos definindo 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 seqüencialmente quando a função for chamada. A palavra int indica que esta função retorna um inteiro. O que significa este retorno será visto posteriormente, quando estudarmos um pouco mais detalhadamente as funções do C. A última linha do programa, **return(0)**; , indica o número inteiro que está sendo retornado pela função, no caso o número 0.

A única coisa que o programa *realmente* faz é chamar a função **printf()**, passando a string (uma string é uma seqüência de caracteres, como veremos brevemente) **"Ola! Eu estou vivo!\n"** como argumento. É por causa do uso da função **printf()** pelo programa que devemos incluir o arquivo- cabeçalho **stdio.h** . A função **printf()** neste caso irá apenas colocar a string na tela do computador. O **n** é uma constante chamada de *constante barra invertida*. No caso, o **n** é a constante barra invertida de "new line" e ele é interpretado como um comando de mudança de linha, isto é, após imprimir *Ola! Eu estou vivo!* o cursor passará para a próxima linha. É importante observar também que os *comandos* do C terminam com **;** .

Podemos agora tentar um programa mais complicado:

```
#include <stdio.h>
int main ()
{
int Dias;          /* Declaracao de Variaveis */
float Anos;
printf ("Entre com o número de dias: "); /* Entrada de Dados */
scanf ("%d",&Dias);
Anos=Dias/365.25;      /* Conversao Dias->Anos */
printf ("\n\n%d dias equivalem a %f anos.\n",Dias,Anos);
return(0);
}
```

Vamos entender como o programa acima funciona. São declaradas duas variáveis chamadas **Dias** e **Anos**. A primeira é um **int** (inteiro) e a segunda um **float** (ponto flutuante). As variáveis declaradas como ponto flutuante existem para armazenar números que possuem casas decimais, como 5,1497.

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 inteira **Dias**. Para tanto usamos a função **scanf()**. A string **"%d"** diz à função que iremos ler um inteiro. O segundo parâmetro passado à função diz que o dado lido deverá ser armazenado na variável **Dias**. É importante ressaltar a necessidade de se colocar um **&** antes do nome da variável a ser lida quando se usa a função **scanf()**. O motivo disto só ficará claro mais tarde. 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 **Anos** o valor de **Dias** dividido por 365.25 (365.25 é uma constante ponto flutuante 365,25). Como **Anos** é uma variável **float** o compilador fará uma conversão automática entre os tipos das variáveis (veremos isto com detalhes mais tarde).

A segunda chamada à função **printf()** tem três argumentos. A string **"\n\n%d dias equivalem a %f anos.\n"** diz à função para pular duas linhas, colocar um inteiro na tela, colocar a mensagem **" dias equivalem a "**, colocar um valor **float** na tela, colocar a mensagem **" anos."** e pular outra linha. Os outros parâmetros são as variáveis, **Dias** e **Anos**, das quais devem ser lidos os valores do inteiro e do **float**, respectivamente.

AUTO AVALIAÇÃO

1 - Veja como você está. O que faz o seguinte programa?

```
#include <stdio.h>
int main()
{
  int x;
  scanf("%d",&x);
  printf("%d",x);
  return(0);
}                }
```

2 - Compile e execute os programas desta página

Introdução às Funções

Uma função é um bloco de código de programa que pode ser usado diversas vezes em sua execução. O uso de funções permite que o programa fique mais legível, mais bem estruturado. Um programa em C consiste, no fundo, de várias funções colocadas juntas.

Abaixo o tipo mais simples de função:

```
#include <stdio.h>
int mensagem ()  /* Funcao simples: so imprime Ola! */
{
        printf ("Ola! ");
        return(0);
}
int main ()
{
        mensagem();
        printf ("Eu estou vivo!\n");
        return(0);
}
```

Este programa terá o mesmo resultado que o primeiro exemplo da seção anterior. O que ele faz é definir uma função **mensagem()** que coloca uma string na tela e retorna 0. Esta função é chamada a partir de **main()** , que, como já vimos, também é uma função. A diferença fundamental entre main e as demais funções do problema é que main é uma função especial, cujo diferencial é o fato de ser a primeira função a ser executada em um programa.

- 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 outro exemplo simples de função com argumentos:

```
 #include <stdio.h>
int square (int x)            /* Calcula o quadrado de x */
{
  printf ("O quadrado e %d",(x*x));
        return(0);
}
int main ()
{
   int num;
   printf ("Entre com um numero: ");
   scanf ("%d",&num);
   printf ("\n\n");
   square(num);
          return(0);
}
```

Na definição de **square()** 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 à 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 **square()** é copiada para a variável **x**. Dentro de **square()** trabalha-se apenas com **x**. Se mudarmos o valor de **x** dentro de **square()** o valor de **num** na função **main()** permanece inalterado.

Vamos dar um exemplo de função de mais de uma variável. Repare que, neste caso, os argumentos são separados por vírgula e que deve-se explicitar o tipo de cada um dos argumentos, um a um. Note, também, que os argumentos passados para a função não necessitam ser todos variáveis porque mesmo sendo constantes serão copiados para a variável de entrada da função.

```
 #include <stdio.h>

int mult (float a, float b,float c)       /* Multiplica 3 numeros */
{
    printf ("%f",a*b*c);
          return(0);
}

 int main ()
 {
   float x,y;
   x=23.5;
```

```
    y=12.9;
    mult (x,y,3.87);
          return(0);
}
```

- Retornando valores

Muitas vezes é necessário fazer com que uma função retorne um valor. As funções que vimos até aqui estavam retornando o número 0. 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**. Sabendo disto fica fácil fazer uma função para multiplicar dois inteiros e que retorna o resultado da multiplicação. Veja:

```
#include <stdio.h>
int prod (int x,int y)
{
    return (x*y);
}
int main ()
{
    int saida;
    saida=prod (12,7);
    printf ("A saida e: %d\n",saida);
    return(0);
}
```

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 o 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 e, neste curso, este valor será sempre especificado.

Com relação à função main, o retorno sempre será inteiro. Normalmente faremos a função main retornar um zero quando ela é executada sem qualquer tipo de erro.

Mais um exemplo de função, que agora recebe dois floats e também retorna um float::

```
#include <stdio.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);
    return(0);
}
```

- Forma geral

Apresentamos aqui a forma geral de uma função:

tipo_de_retorno nome_da_função (lista_de_argumentos)
{
* código_da_função*
}

AUTO AVALIAÇÃO

Veja como você está. Escreva uma função que some dois inteiros e retorne o valor da soma.

Introdução Básica às Entradas e Saídas

- Caracteres

Os caracteres são um tipo de dado: o **char**. O C trata os caracteres ('a', 'b', 'x', etc ...) como sendo variáveis de um *byte* (8 *bit*s). Um *bit* é a menor unidade de armazenamento de informações em um computador. Os inteiros (**int**s) têm um número maior de *bytes*. Dependendo da implementação do compilador, eles podem ter 2 *bytes* (16 *bits*) ou 4 *bytes* (32 *bits*). Isto será melhor explicado na aula 3. 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 apóstrofes. Veja um exemplo de programa que usa caracteres:

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

No programa acima, **%c** indica que **printf()** deve colocar um caractere na tela. Como vimos 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>
int main ()
{
  char Ch;
  Ch='D';
  printf ("%d",Ch);  /* Imprime o caracter como inteiro */
  return(0);
}
```

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

Muitas vezes queremos ler um caractere fornecido pelo usuário. Para isto as funções mais usadas, quando se está trabalhando em ambiente DOS ou Windows, são **getch()** e **getche()**. Ambas retornam o caractere pressionado. **getche()** imprime o caractere na tela antes de retorná-lo e **getch()** apenas retorna o caractere pressionado sem imprimí-lo na tela. Ambas as funções podem ser encontradas no arquivo de cabeçalho **conio.h**. Geralmente estas funções **não estão disponíveis em ambiente Unix** (compiladores cc e gcc), pois não fazem

parte do padrão ANSI. Podem ser substituídas pela função scanf(), porém sem as mesmas funcionalidades. Eis um exemplo que usa a função **getch(),** e seu correspondente em ambiente Unix:

```
#include <stdio.h>
#include <conio.h>
/* Este programa usa conio.h . Se você não tiver a conio, ele não funcionará no Unix */
int main ()
{
        char Ch;
        Ch=getch();
        printf ("Voce pressionou a tecla %c",Ch);
        return(0);
}
```

Equivalente ANSI-C para o ambiente Unix do programa acima, sem usar getch():

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

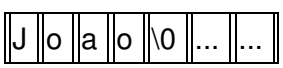
A principal diferença da versão que utiliza getch() para a versão que não utiliza getch() é que no primeiro caso o usuário simplesmente aperta a tecla e o sistema lê diretamente a tecla pressionada. No segundo caso, é necessário apertar também a tecla <ENTER>. **Lembre-se que, se você quiser manter a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

- 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 terminador nulo também pode ser escrito usando a convenção de barra invertida do C como sendo '\0'. Embora o assunto vetores seja discutido posteriormente, veremos aqui os fundamentos necessários para que possamos utilizar as strings. 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 caractere para ser o terminador 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 7 posições e coloquemos a palavra João nela. Teremos:



No caso acima, as duas células não usadas têm valores indeterminados. Isto acontece porque o C *não* inicializa variáveis, cabendo ao programador esta tarefa. Portanto as únicas células que são inicializadas são as que contêm os caracteres 'J', 'o', 'a', 'o' e '\0' .

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>
int main ()
{
   char string[100];
   printf ("Digite uma string: ");
   gets (string);
   printf ("\n\nVoce digitou %s",string);
   return(0);
}
```

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

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

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

Por quê se está acessando a segunda letra e não a primeira? 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 segunda letra da string "Joao", apresentada acima. Em seguida, ele mudará esta letra e apresentará a string no final.

```
#include <stdio.h>
int main()
{
   char str[10] = "Joao";
   printf("\n\nString: %s", str);
   printf("\nSegunda letra: %c", str[1]);
   str[1] = 'U';
   printf("\nAgora a segunda letra eh: %c", str[1]);
   printf("\n\nString resultante: %s", str);
       return(0);
}
```

Nesta string, o terminador nulo está na posição 4. Das posições 0 a 4, sabemos que temos caracteres válidos, e portanto podemos escrevê-los. Note a forma como inicializamos a string **str** com os caracteres 'J' 'o' 'a' 'o' e '\0' simplesmente declarando char str[10] = "Joao". Veremos, posteriormente que "Joao" (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.

No programa acima, **%s** indica que **printf()** deve colocar uma string na tela. Vamos agora fazer uma abordagem inicial às duas funções que já temos usado para fazer a entrada e saída.

- 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 posição 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 ("Um caractere %c e um inteiro %d",'D',120) -> "Um caractere D e um inteiro 120"
printf ("%s e um exemplo","Este") -> "Este e um exemplo"
printf ("%s%d%%","Juros de ",10) -> "Juros de 10%"

Maiores detalhes sobre a função **printf()** (incluindo outros códigos de controle) serão vistos posteriormente, mas podem ser consultados de antemão pelos interessados.

- scanf

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

scanf (string-de-controle,lista-de-argumentos);

Usando a função **scanf()** podemos pedir dados ao usuário. Um exemplo de uso, pode ser visto acima. 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. É impossível justificar isto agora, mas veremos depois a razão para este procedimento. Maiores detalhes sobre a função **scanf()** serão vistos posteriormente, mas podem ser consultados de antemão pelos interessados.

AUTO AVALIAÇÃO

Veja como você está:

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

Introdução a Alguns Comandos de Controle de Fluxo

Os comandos de controle de fluxo são aqueles que permitem ao programador alterar a sequência de execução do programa. Vamos dar uma breve introdução a dois comandos de controle de fluxo. Outros comandos serão estudados posteriormente.

- if

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

if (condição) declaração;

A condição do comando **if** é uma expressão que será avaliada. Se o resultado for zero a declaração não será executada. Se o resultado for qualquer coisa diferente de zero a declaração será executada. 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>
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");
            return (0);
}
```

No programa acima a expressão **num>10** é avaliada e retorna um valor diferente de zero, se verdadeira, e zero, se falsa. No exemplo, se num for maior que 10, será impressa a frase: "O número e maior que 10". Repare que, se o número for igual a 10, estamos executando dois comandos. Para que isto fosse possível, tivemos que agrupa-los em um bloco que se inicia logo após a comparação e termina após o segundo printf. Repare também que quando queremos testar igualdades usamos o operador **==** e não **=**. Isto porque o operador **=** representa *apenas* uma atribuição. Pode parecer estranho à primeira vista, mas se escrevêssemos

```
if (num=10) ...       /* Isto esta 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 modificado e fazendo com que a declaração fosse executada sempre. Este problema gera erros frequentes entre iniciantes e, portanto, muita atenção deve ser tomada.

Os operadores de comparação são:

== (igual), **!=** (diferente de),

> (maior que)**, <** (menor que)**,**

>= (maior ou igual**), <=** (menor ou igual)**.**

- for

O loop (laço) **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. 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 loop **for** é ver de que maneira ele funciona "por dentro". O loop **for** é equivalente a se fazer o seguinte:

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

Podemos ver que o **for** executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, 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 na tela:

```
#include <stdio.h>
int main ()
{
   int count;
    for (count=1;count<=100;count=count+1)
          printf ("%d ",count);
    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>
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 de caracteres */
   }
   printf("\nNumero de caracteres c = %d", cont);
```

```
   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.

Mais um exemplo, agora envolvendo caracteres:

```
/* Este programa imprime o alfabeto: letras maiúsculas  */
#include <stdio.h>
int main()
{
   char letra;
   for(letra = 'A' ; letra <= 'Z' ; letra =letra+1)
   printf("%c ", letra);
}
```

Este programa funciona porque as letras maiúsculas de A a Z possuem código inteiro sequencial.

AUTO AVALIAÇÃO

Veja como você está.

a) Explique porque está **errado** fazer

```
if (num=10) ...
```

O que irá acontecer?

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

c) 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.

Comentários

Como já foi dito, o uso de comentários torna o código do programa mais fácil de se entender. Os comentários do C devem começar com /* e terminar com */. O C padrão não permite comentários aninhados (um dentro do outro), mas alguns compiladores os aceitam.

Palavras Reservadas do C

Todas as linguagens de programação têm palavras reservadas. As palavras reservadas não podem ser usadas a não ser nos seus propósitos originais, isto é, não podemos declarar funções ou variáveis com os mesmos nomes. 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. Veremos o significado destas palavras chave à medida em que o curso for progredindo:

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

Aula 3 - VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES

Nomes de Variáveis

As variáveis no C podem ter qualquer nome se duas condições forem satisfeitas: o nome deve começar com uma letra ou sublinhado (_) e os caracteres subsequentes devem ser letras, números ou sublinhado (_). Há apenas mais duas restrições: 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.

Dicas quanto aos nomes de variáveis...

- É uma prática tradicional do C, usar letras minúsculas para nomes de variáveis e maiúsculas para nomes de constantes. Isto facilita na hora da leitura do código;
- Quando se escreve código usando nomes de variáveis em português, evita-se possíveis conflitos com nomes de rotinas encontrados nas diversas bibliotecas, que são em sua maioria absoluta, palavras em inglês.

Os Tipos do C

O C tem 5 tipos básicos: **char**, **int**, **float**, **void, double**. Destes não vimos ainda os dois últimos: O **double** é o ponto flutuante duplo e pode ser visto como um ponto flutuante com muito mais precisão. O void é o tipo vazio, ou um "tipo sem tipo". A aplicação deste "tipo" será vista posteriormente.

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 modificadores podem ser aplicados a inteiros. A intenção é que **short** e **long** devam prover tamanhos diferentes de inteiros onde isto for prático. Inteiros menores (**short**) ou maiores (**long**). **int** normalmente terá o tamanho natural para uma determinada máquina. Assim, numa máquina de 16 bits, **int** provavelmente terá 16 bits. Numa máquina de 32, **int** deverá ter 32 bits. Na verdade, cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que **shorts ints** e **ints** devem ocupar pelo menos 16 bits, **longs ints** pelo menos 32 bits, e **short int** não pode ser maior que **int**, que não pode ser maior que **long int**. O modificador **unsigned** serve para especificar variáveis sem sinal. Um **unsigned int** será um inteiro que assumirá apenas valores positivos. A seguir estão listados os tipos de dados permitidos e seu valores máximos e mínimos em um compilador típico para um hardware de 16 bits. Também nesta tabela está especificado o formato que deve ser utilizado para ler os tipos de dados com a função scanf():

Tipo	Num de bits	Formato para leitura com scanf	Intervalo	
			Inicio	Fim
char	8	%c	-128	127
unsigned char	8	%c	0	255
signed char	8	%c	-128	127
int	16	%i	-32.768	32.767
unsigned int	16	%u	0	65.535
signed int	16	%i	-32.768	32.767
short int	16	%hi	-32.768	32.767
unsigned short int	16	%hu	0	65.535
signed short int	16	%hi	-32.768	32.767
long int	32	%li	-2.147.483.648	2.147.483.647
signed long int	32	%li	-2.147.483.648	2.147.483.647
unsigned long int	32	%lu	0	4.294.967.295
float	32	%f	3,4E-38	3.4E+38
double	64	%lf	1,7E-308	1,7E+308
long double	80	%Lf	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.

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 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 count;
float pi;
```

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

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>
int contador;
int func1(int j) {
/* aqui viria o código da funcao
                        ...
          */
}
int main()
{
  char condicao;
  int i;
  for (i=0; i<100; i=i+1)
  {          /* Bloco do for */
    float f2;
    /* etc ...
                        ...
              */
    func1(i);
  }
  /* etc  ...  */
            return(0);
}
```

A variável *contador* é 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.

Podemos inicializar variáveis no momento de sua declaração. Para fazer isto podemos usar a forma geral

tipo_da_variável nome_da_variável = constante;

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 count=0;
float pi=3.141;
```

Ressalte-se novamente que, em C, uma variável tem que ser declarada no início de um bloco de código. Assim, o programa a seguir não é válido em C (embora seja válido em C++).

```
int  main()
{
      int i;
      int j;
      j = 10;
      int k = 20;  /* Esta declaracao de variável não é válida, pois não está sendo feita no início do bloco */
            return(0);
}
```

AUTO AVALIAÇÃO

Veja como você está:

Escreva um programa que declare uma variável inteira global e atribua o valor 10 a ela. Declare outras 5 variáveis inteiras locais ao programa principal e atribua os valores 20, 30, ..., 60 a elas. Declare 6 variáveis caracteres e atribua a elas as letras c, o, e, l, h, a . Finalmente, o programa deverá imprimir, usando todas as variáveis declaradas:

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

O animal contido nas variáveis caracteres e' a coelha

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...

- 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

- Constantes hexadecimais e octais

Muitas vezes precisamos inserir constantes hexadecimais (base dezesseis) ou octais (base oito) no nosso 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)

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

- 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'**.

- Constantes de barra invertida

O C utiliza, para nos facilitar a tarefa de programar, vários códigos chamados códigos de barra invertida. Estes são caracteres que podem ser usados como qualquer outro. Uma lista com alguns dos códigos de barra invertida é dada a seguir:

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

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". O operador % fornece o resto da divisão de dois inteiros.

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 após 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 à 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) ...        /* Expressão 2 */
```

A expressão 1 é válida, pois quando fazemos **z=1.5** ela retorna 1.5, que é passado adiante, fazendo y = 1.5 e posteriormente x = 1.5. A expressão 2 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.

AUTO AVALIAÇÃO

Veja como você está:

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--);
```

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>
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);
```

```
    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>
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);
}
```

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>
int main()
{
    int i;
    for(i=1; i<=100; i++)
        if(!(i%2)) printf("%d ",i);     /* o operador de resto dará falso (zero) */
}                                       /* quando usada c/ número par. Esse resultado*/
                                        /* é invertido pelo ! */
```

- 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.

Aula 4 - 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.

O Comando if

Já introduzimos o comando **if**. Sua forma geral é:

if (condição) declaração;

A expressão, na condição, será avaliada. Se ela for zero, a declaração não será executada. Se a condição for diferente de zero a declaração será executada. Aqui reapresentamos o exemplo de um uso do comando **if** :

```
#include <stdio.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");
	return(0);
}
```

- 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 diferente de zero a declaração 1 será executada. Se for zero 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>
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");
	}
	return(0);
}
```

- 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;

A estrutura acima funciona da seguinte maneira: o programa começa a testar as condições começando pela 1 e continua a testar até que ele ache uma expressão cujo resultado dê diferente de zero. Neste caso ele executa a declaração correspondente. Só uma declaração será executada, ou seja, só será executada a declaração equivalente à *primeira* condição que der diferente de zero. A última declaração (default) é a que será executada no caso de todas as condições darem zero e é opcional.

Um exemplo da estrutura acima:

```
#include <stdio.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");
        return(0);
}
```

- 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. Isto quer dizer que teremos as seguintes expressões:

```
int num;
if (num!=0) ....
if (num==0) ....
        for (i = 0; string[i] != '\0'; i++)
```

equivalem a

```
int num;
if (num) ....
if (!num) ....
        for (i = 0; string[i]; i++)
```

Isto quer dizer que podemos simplificar algumas expressões simples.

- 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>
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.");
         }
    }
        return(0);
}
```

- O Operador ?

Uma expressão como:

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

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

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

De uma maneira geral expressões do tipo:

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

podem ser substituídas por:

condição?expressão_1:expressão_2;

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>
int main()
{
    int index = 0, contador;
    char letras[5] = "Joao";
    for (contador=0; contador < 1000; contador++)
    {
            printf("\n%c",letras[index]);
            (index==3) ? index=0: ++index;
    }
}
```

O nome Joao é escrito na tela verticalmente até a variável contador determinar o término do programa. Enquanto isto a variável index assume os valores 0, 1, 2, 3, , 0, 1, ... progressivamente.

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. Mais uma vez vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. 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 eu recomendo cuidado. Veremos agora um exemplo do comando **switch**:

```
#include <stdio.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\nO numero nao e nem 9 nem 10 nem 11.\n");
    }
        return(0);
}
```

AUTO AVALIAÇÃO

Veja como você está.

Escreva um programa que pede para o usuário entrar um número correspondente a um dia da semana e que então apresente na tela o nome do dia. utilizando o comando switch.

O Comando for

for é a primeira de uma série de três estruturas para se trabalhar com loops 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.

Como já foi dito, o loop **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:

for (inicialização;condição;incremento) declaração;

O melhor modo de se entender o loop **for** é ver como ele funciona "por dentro". O loop **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 incondicionalmente 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. Um ponto importante é que podemos omitir qualquer um dos elementos do **for**, isto é, se não quisermos uma inicialização poderemos omiti-la. Abaixo vemos um programa que coloca os primeiros 100 números inteiros na tela:

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

Note que, no exemplo acima, há uma diferença em relação ao exemplo anterior. O incremento da variável **count** é feito usando o operador de incremento que nós agora já conhecemos. Esta é a forma usual de se fazer o incremento (ou decremento) em um loop **for**.

O **for** na linguagem C é bastante flexível. Temos acesso à inicialização, à condição e ao incremento. Qualquer uma destas partes do for pode ser uma expressão qualquer do C, desde que ela seja válida. Isto nos permite fazer o que quisermos com o comando. As três formas do for abaixo são válidas:

```
for ( count = 1; count < 100 ; count++) { ... }
for (count = 1; count < NUMERO_DE_ELEMENTOS ; count++) { ... }
for (count = 1; count < BusqueNumeroDeElementos() ; count+=2) { ... }
etc ...
```

Preste atenção ao último exemplo: o incremento está sendo feito de dois em dois. Além disto, no teste está sendo utilizada uma função (BusqueNumeroDeElementos()) que retorna um valor que está sendo comparado com count.

- O loop infinito

O loop infinito tem a forma

for (inicialização; ;incremento) declaração;

Este loop chama-se loop 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 loop como este usamos o comando **break**. O comando **break** vai quebrar o loop 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 usuario aperte uma tecla sinalizadora de final (um FLAG). O nosso FLAG será a letra 'X'. Repare que tivemos que usar dois scanf() dentro do for. Um busca o caractere que foi digitado e o outro busca o outro caracter digitado na seqüência, que é o caractere correspondente ao <ENTER>.

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

```
  int Count;
  char ch;
  printf(" Digite uma letra - <X para sair> ");
  for (Count=1;;Count++)
  {
    scanf("%c", &ch);
    if (ch == 'X') break;
    printf("\nLetra: %c \n",ch);
    scanf("%c", &ch);
  }
  return(0);
}
```

- O loop sem conteúdo

Loop 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

```
#include <stdio.h>
int main ()
{
        long int i;
        printf("\a");                /* Imprime o caracter de alerta (um beep) */
        for (i=0; i<10000000; i++);  /* Espera 10.000.000 de iteracoes */
        printf("\a");               /* Imprime outro caracter de alerta */
        return(0);
}
```

faz isto.

AUTO AVALIAÇÃO

Veja como você está.

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.

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 loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um loop sem conteúdo. Vamos ver um exemplo do uso do **while**. O programa abaixo é executado enquanto i for menor que 100. Veja que ele seria implementado mais naturalmente com um for ...

```
#include <stdio.h>
int main ()
{
        int i = 0;
        while ( i < 100)
        {
        printf(" %d", i);
                i++;
        }
        return(0);
}
```

O programa abaixo espera o usuário digitar a tecla 'q' e só depois finaliza:

```
#include <stdio.h>
int main ()
{
        char Ch;
        Ch='\0';
        while (Ch!='q')
    {
        scanf("%c", &Ch);
    }
        return(0);
}
```

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** "por dentro":

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 extrutura **do-while** é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido, conforme apresentado abaixo:

```
#include <stdio.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;
    }
        return(0);
}
```

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* loop (como no caso do **for**, do **while** ou do **do while**). O **break** faz com que a execução do programa continue na primeira linha seguinte ao loop 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)
{
        count=1;
        for(;;)
        {
                printf("%d", count);
                count++;
                if(count==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 observaçao é o fato que um break usado dentro de uma declaraçao switch afetará somente os dados relacionados com o switch e nao qualquer outro laço em que o switch estiver.

O Comando continue

O comando **continue** pode ser visto como sendo o oposto do **break**. Ele só funciona dentro de um loop. Quando o comando **continue** é encontrado, o loop pula para a próxima iteração, sem o abandono do loop, ao contrário do que acontecia no comando **break**. O programa abaixo exemplifica o uso do *continue*:

```
#include <stdio.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;
        }
    }
return(0);
}
```

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

Aula 5 - MATRIZES E STRINGS

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 contígua.

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 sobreescritas ou de ver o computador travar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

```
#include <stdio.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]);
    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 a 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.

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>
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 em '\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.

- 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>
int main ()
{
        char string[100];
        printf ("Digite o seu nome: ");
        gets (string);
        printf ("\n\n Ola %s",string);
        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.

- 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>
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);
        return(0);
}
```

- 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>
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);
        return(0);
}
```

- 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>
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);
}
```

- 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>
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.");
        return(0);
}
```

AUTO AVALIAÇÃO

Veja como você está.

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.

Matrizes

- 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>
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++;
    }
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.

- 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>
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]);
    return(0);
}
```

- 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.

- Inicialização

Podemos inicializar matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz como 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.

- 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.

AUTO AVALIAÇÃO

Veja como você está.

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

```
# include <stdio.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");
   }
   return(0);
}
```

Aula 6 –FUNÇÕES

A Função

Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se não as tivéssemos, os programas teriam que ser curtos e de pequena

complexidade. Para fazermos programas grandes e complexos temos de construí-los bloco a bloco.

Uma função no C tem a seguinte forma geral:

tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
corpo_da_função
}

O tipo-de-retorno é o tipo de variável que a função vai retornar. O default é o tipo **int**, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro. A declaração de parâmetros é uma lista com a seguinte forma geral:

tipo nome1, tipo nome2, ... , tipo nomeN

Repare que o tipo deve ser especificado para cada uma das N variáveis de entrada. É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no tipo-de-retorno).

O corpo da função é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas.

O Comando return

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

return valor_de_retorno; ou *return;*

Digamos que uma função está sendo executada. Quando se chega a uma declaração **return** a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor. É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.

Uma função pode ter mais de uma declaração **return**. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração **return**. Abaixo estão dois exemplos de uso do **return**:

```
#include <stdio.h>
int Square (int a)
{
        return (a*a);
}
int main ()
{
        int num;
        printf ("Entre com um numero: ");
        scanf ("%d",&num);
        num=Square(num);
        printf ("\n\nO seu quadrado vale: %d\n",num);
        return 0;
}

#include <stdio.h>
int EPar (int a)
{
```

```
        if (a%2)        /* Verifica se a e divisivel por dois */
    return 0;       /* Retorna 0 se nao for divisivel */
        else
    return 1;       /* Retorna 1 se for divisivel */
}
int main ()
{
        int num;
        printf ("Entre com numero: ");
        scanf ("%d",&num);
        if (EPar(num))
        printf ("\n\nO numero e par.\n");
        else
        printf ("\n\nO numero e impar.\n");
        return 0;
}
```

É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições, ou mesmo para que estes valores participem de expressões. Mas *não* podemos fazer:

```
func(a,b)=x;    /* Errado! */
```

No segundo exemplo vemos o uso de mais de um **return** em uma função.

Fato importante: se uma função retorna um valor você *não precisa aproveitar* este valor. Se você não fizer nada com o valor de retorno de uma função ele será descartado. Por exemplo, a função **printf()** retorna um inteiro que nós nunca usamos para nada. Ele é descartado.

AUTO AVALIAÇÃO

Veja como você está. Escreva a função 'EDivisivel(int a, int b)' (tome como base EPar(int a)). A função deverá retornar 1 se o resto da divisão de a por b for zero. Caso contrário, a função deverá retornar zero.

Protótipos de Funções

Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função **main()**. Isto é, as funções estão fisicamente antes da função **main()**. Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função **main()**, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto as funções foram colocadas antes da função **main()**: quando o compilador chegasse à função **main()** ele já teria compilado as funções e já saberia seus formatos.

Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?

A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

tipo_de_retorno nome_da_função (declaração_de_parâmetros);

onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:

```
#include <stdio.h>
float Square (float a);
int main ()
{
        float num;
        printf ("Entre com um numero: ");
        scanf ("%f",&num);
        num=Square(num);
        printf ("\n\nO seu quadrado vale: %f\n",num);
        return 0;
}
float Square (float a)
{
        return (a*a);
}
```

Observe que a função **Square()** está colocada depois de **main()**, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.

Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. É bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!

O Tipo void

Agora vamos ver o único tipo da linguagem C que não detalhamos ainda: o **void**. Em inglês, **void** quer dizer vazio e é isto mesmo que o **void** é. Ele nos permite fazer funções que não retornam nada e funções que não têm parâmetros! Podemos agora escrever o protótipo de uma função que não retorna nada:

void nome_da_função (declaração_de_parâmetros);

Numa função, como a acima, não temos valor de retorno na declaração **return**. Aliás, neste caso, o comando **return** não é necessário na função.

Podemos, também, fazer funções que não têm parâmetros:

tipo_de_retorno nome_da_função (void);

ou, ainda, que não tem parâmetros e não retornam nada:

void nome_da_função (void);

Um exemplo de funções que usam o tipo **void**:

```
#include <stdio.h>
void Mensagem (void);
int main ()
{
        Mensagem();
        printf ("\tDiga de novo:\n");
        Mensagem();
        return 0;
}
void Mensagem (void)
{
        printf ("Ola! Eu estou vivo.\n");
}
```

Se quisermos que a função retorne algo, devemos usar a declaração **return**. Se não quisermos, basta declarar a função como tendo tipo-de-retorno **void**. Devemos lembrar agora que a função **main()** *é* uma função e como tal devemos tratá-la. O compilador acha que a função **main()** deve retornar um inteiro. Isto pode ser interessante se quisermos que o sistema operacional receba um valor de retorno da função **main()**. Se assim o quisermos, devemos nos lembrar da seguinte convenção: se o programa retornar zero, significa que ele terminou normalmente, e, se o programa retornar um valor diferente de zero, significa que o programa teve um término anormal. Se não estivermos interessados neste tipo de coisa, basta declarar a função main como retornando **void**.

As duas funções **main()** abaixo são válidas:

```
main (void)
{
        ....
        return 0;
}

void main (void)
{
        ....
}
```

A primeira forma é válida porque, como já vimos, as funções em C têm, por padrão, retorno inteiro.. Alguns compiladores reclamarão da segunda forma de main, dizendo que main sempre deve retornar um inteiro. Se isto acontecer com o compilador que você está utilizando, basta fazer main retornar um inteiro.

Passagem de parâmetros por valor e passagem por referência

Já vimos que, na linguagem C, quando chamamos uma função os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é denominado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios parâmetros. Veja o exemplo abaixo:

```
#include <stdio.h>
float sqr (float num);
void main ()
{
        float num,sq;
        printf ("Entre com um numero: ");
```

```
        scanf ("%f",&num);
        sq=sqr(num);
        printf ("\n\nO numero original e: %f\n",num);
        printf ("O seu quadrado vale: %f\n",sq);
}

float sqr (float num)
{
        num=num*num;
        return num;
}
```

No exemplo acima o parâmetro formal **num** da função **sqr()** sofre alterações dentro da função, mas a variável **num** da função **main()** permanece inalterada: é uma chamada por valor.

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).

O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem termos que nos preocupar em estar alterando os valores dos parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porque podemos querer mudar os valores dos parâmetros fora da função também. O C++ tem um recurso que permite ao programador fazer chamadas por referência. Há entretanto, no C, um recurso de programação que podemos usar para simular uma chamada por referência.

Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*. Os ponteiros são a "referência" que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um **&** na frente das variáveis que estivermos passando para a função. Veja um exemplo:

```
#include <stdio.h>
void Swap (int *a,int *b);
void main (void)
{
        int num1,num2;
        num1=100;
        num2=200;
        Swap (&num1,&num2);
        printf ("\n\nEles agora valem %d  %d\n",num1,num2);
}
void Swap (int *a,int *b)
{
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
```

Não é muito difícil. O que está acontecendo é que passamos para a função Swap o endereço das variáveis num1 e num2. Estes endereços são copiados nos ponteiros a e b. Através do operador * estamos acessando o conteúdo apontado pelos ponteiros e modificando-

o. Mas, quem é este conteúdo? Nada mais que os valores armazenados em num1 e num2, que, portanto, estão sendo modificados!

Espere um momento... será que nós já não vimos esta estória de chamar uma função com as variáveis precedidas de **&**? Já! É assim que nós chamamos a função **scanf()**. Mas porquê? Vamos pensar um pouco. A função **scanf()** usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela! Não é para isto mesmo que ela é feita? Ela lê variáveis para nós e portanto precisa alterar seus valores. Por isto passamos para a função o endereço da variável a ser modificada!

AUTO AVALIAÇÃO

Veja como você está

Escreva uma função que receba duas variáveis inteiras e "zere" o valor das variáveis. Use o que você aprendeu nesta página para fazer a implementação

Vetores como Argumentos de Funções

Quando vamos passar um vetor como argumento de uma função, podemos declarar a função de três maneiras equivalentes. Seja o vetor:

```
int matrx [50];
```

e que queiramos passá-la como argumento de uma função **func()**. Podemos declarar **func()** das três maneiras seguintes:

```
void func (int matrx[50]);
void func (int matrx[]);
void func (int *matrx);
```

Nos três casos, teremos dentro de **func()** um **int*** chamado **matrx**. Ao passarmos um vetor para uma função, na realidade estamos passando um ponteiro. Neste ponteiro é armazenado o endereço do primeiro elemento do vetor. Isto significa que não é feita uma cópia, elemento a elemento do vetor. Isto faz com que possamos alterar o valor dos elementos do vetor dentro da função.

Outras Questões

Uma função, como foi dito anteriormente, é um bloco de construção muito útil. No C as funções são flexíveis. A flexibilidade dá poder, mas exige cuidado.

Funções devem ser implementadas, quando possível, da maneira mais geral possível. Isto as torna mais fáceis de serem reutilizadas e entendidas. Evite, sempre que possível, funções que usem variáveis globais.

Se houver uma rotina que deve ser o mais veloz possível, seria bom implementá-la sem nenhuma (ou com o mínimo de) chamadas a funções, porque uma chamada a uma função consome tempo e memória.

Um outro ponto importante é que, como já sabemos um bocado a respeito de funções, quando formos ensinar uma das funções das bibliotecas do C vamos mostrar, em primeiro

lugar, o seu protótipo. Quem entendeu tudo que foi ensinado nesta parte sobre funções pode retirar inúmeras informações de um protótipo (tipo de retorno, nome da função, tipo dos argumentos, passagem por valor ou passagem por referência).

Sugiro que neste ponto, o leitor leia um arquivo-cabeçalho como, por exemplo o **stdio.h** ou o **string.h**. É um bom treino. Estes arquivo podem ser encontrados no diretório apropriado do compilador que você estiver utilizando (geralmente o subdiretório include do diretório onde você instalou o compilador).

Aula 7 - DIRETIVAS DE COMPILAÇÃO

As Diretivas de Compilação

O pré-processador C é um programa que examina o programa fonte escrito em C e executa certas modificações nele, baseado nas *Diretivas de Compilação*. As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, que é executado pelo compilador antes da execução do processo de compilação propriamente dito.

Portanto, o pré-processador modifica o programa fonte, entregando para o compilador um programa modificado. Todas as diretivas de compilação são iniciadas pelo caracter #. As diretivas podem ser colocadas em qualquer parte do programa. Já vimos, e usamos muito, a diretiva **#include**. Sabemos que ela não *gera* código mas diz ao compilador que ele deve incluir um arquivo externo na hora da compilação. As diretivas do C são identificadas por começarem por **#**. As diretivas que estudaremos são definidas pelo padrão ANSI:

#if	*#ifdef*	*#ifndef*
#else	*#elif*	*#endif*
#include	*#define*	*#undef*

Procuraremos ser breves em suas descrições...

A Diretiva include

A diretiva **#include** já foi usada durante o nosso curso diversas vezes. Ela diz ao compilador para incluir, na hora da compilação, um arquivo especificado. Sua forma geral é:

#include "nome_do_arquivo"

ou

#include <nome_do_arquivo>

A diferença entre se usar **" "** e **< >** é somente a ordem de procura nos diretórios pelo arquivo especificado. Se você quiser informar o nome do arquivo com o caminho completo, ou se o arquivo estiver no diretório de trabalho, use **" "**. Se o arquivo estiver nos caminhos de procura pré-especificados do compilador, isto é, se ele for um arquivo do próprio sistema (como é o caso de arquivos como **stdio.h**, **string.h,** etc...) use **< >**.

Observe que não há ponto e vírgula após a diretiva de compilação. Esta é uma característica importante de todas as diretivas de compilação e não somente da diretiva #include

As Diretivas define e undef

A diretiva **#define** tem a seguinte forma geral:

#define nome_da_macro sequência_de_caracteres

Quando você usa esta diretiva, você está dizendo ao compilador para que, toda vez que ele encontrar o nome_da_macro no programa a ser compilado, ele deve substituí-lo pela sequência_de_caracteres fornecida. Isto é muito útil para deixar o programa mais geral. Veja um exemplo:

```
#include <stdio.h>
#define PI 3.1416
#define VERSAO "2.02"
int main ()
{
        printf ("Programa versao %s",VERSAO);
        printf ("O numero pi vale: %f",PI);
        return 0;
}
```

Se quisermos mudar o nosso valor de **PI**, ou da **VERSAO**, no programa acima, basta mexer no início do programa. Isto torna o programa mais flexível. Há quem diga que, em um programa, nunca se deve usar constantes como 10, 3.1416, etc., pois estes são números que ninguém sabe o que significam (muitas pessoas os chamam de "números mágicos"). Ao invés disto, deve-se usar apenas **#define**s. É uma convenção de programação (que deve ser seguida, pois torna o programa mais legível) na linguagem C que as macros declaradas em **#define**s devem ser todas em maiúsculas.

Um outro uso da diretiva #define é o de simplesmente definir uma macro. Neste caso usa-se a seguinte forma geral:

#define nome_da_macro

Neste caso o objetivo não é usar a macro no programa (pois ela seria substituída por nada), mas, sim, definir uma macro para ser usada como uma espécie de flag. Isto quer dizer que estamos definindo um valor como sendo "verdadeiro" para depois podermos testá-lo.

Também é possível definir macros com argumentos. Veja o exemplo a seguir:

```
#define max(A,B) ((A>B) ? (A):(B))
#define min(A,B) ((A<B) ? (A):(B))
...
x = max(i,j);
y = min(t,r);
```

Embora pareça uma chamada de função, o uso de max (ou min) simplesmente substitui, em tempo de compilação, o código especificado. Cada ocorrência de um parâmetro formal (A ou B, na definição) será substituído pelo argumento real correspondente. Assim, a linha de código:

```
x = max(i,j);
```

será substituída pela linha:

```
x = ((i)>(j) ? (i):(j));
```

A linha de código:

x = max(p+q,r+s);

será substituída pela linha:

x = ((p+q)>(r+s) ? (p+q):(r+s));

Isto pode ser muito útil. Verifique que as macros max e min não possuem especificação de tipo. Logo, elas trabalham corretamente para qualquer tipo de dado, enquanto os argumentos passados forem coerentes. Mas isto pode trazer também algumas armadilhas. Veja que a linha

x = max(p++,r++);

será substituída pelo código

x = ((p++)>(r++) ? (p++):(r++));

e em consequência, incrementará o maior valor duas vezes.

Outra armadilha em macros está relacionada com o uso de parênteses. Seja a macro:

#define SQR(X) X*X

Imagine que você utilize esta macro na expressão abaixo:

y = SQR(A+B);

Ao fazer isto, a substituição que será efetuada não estará correta. A expressão gerada será:

y = A+B*A+B;

que obviamente é diferente de (A+B)*(A+B) !

A solução para este problema é incluir parênteses na definição da macro:

#define SQR(X)(X)*(X)

Quando você utiliza a diretiva #define nunca deve haver espaços em branco no identificador. Por exemplo, a macro:

#define PRINT (i) printf(" %d \n", i)

não funcionará corretamente porque existe um espaço em branco entre PRINT e (i). Ao se tirar o espaço, a macro funcionará corretamente e poderá ser utilizada para imprimir o número inteiro i, saltando em seguida para a próxima linha.

A diretiva **#undef** tem a seguinte forma geral:

#undef nome_da_macro

Ela faz com que a macro que a segue seja apagada da tabela interna que guarda as macros.O compilador passa a partir deste ponto a não conhecer mais esta macro.

Aula 8 - Entradas e Saídas Padronizadas

Introdução

O sistema de entrada e saída da linguagem C está estruturado na forma de uma biblioteca de funções . Já vimos algumas destas funções, e agora elas serão reestudadas. Novas funções também serão apresentadas.

Não é objetivo deste curso explicar, em detalhes, todas as possíveis funções da biblioteca de entrada e saída do C. A sintaxe completa destas funções pode ser encontrada no manual do seu compilador. Alguns sistemas trazem um descrição das funções na ajuda do compilador, que pode ser acessada "on line". Isto pode ser feito, por exemplo, no Rhide.

Um ponto importante é que agora, quando apresentarmos uma função, vamos, em primeiro lugar, apresentar o seu protótipo. Você já deve ser capaz de interpretar as informações que um protótipo nos passa. Se não, deve voltar a estudar a aula sobre funções.

Outro aspecto importante, quando se discute a entrada e saída na linguagem C é o conceito de *fluxo*. Seja qual for o dispositivo de entrada e saída (discos, terminais, teclados, acionadores de fitas) que se estiver trabalhando, o C vai enxergá-lo como um fluxo, que nada mais é que um dispositivo lógico de entrada ou saída. Todos os fluxos são similares em seu funcionamento e independentes do dispositivo ao qual estão associados. Assim, as mesmas funções que descrevem o acesso aos discos podem ser utilizadas para se acessar um terminal de vídeo. Todas as operações de entrada e saída são realizadas por meio de fluxos.

Na linguagem C, um arquivo é entendido como um conceito que pode ser aplicado a arquivos em disco, terminais, modens, etc ... Um fluxo é associado a um arquivo através da realização de uma operação de abertura. Uma vez aberto, informações podem ser trocadas entre o arquivo e o programa. Um arquivo é dissociado de um fluxo através de uma operação de fechamento de arquivo.

Lendo e Escrevendo Caracteres

Uma das funções mais básicas de um sistema é a entrada e saída de informações em dispositivos. Estes podem ser um monitor, uma impressora ou um arquivo em disco. Vamos ver os principais comandos que o C nos fornece para isto.

- getche e getch

As funções getch() e getche() não são definidas pelo padrão ANSI. Porém, elas geralmente são incluídas em compiladores baseados no DOS, e se encontram no header file *conio.h*. Vale a pena repetir: são funções comuns apenas para compiladores baseados em DOS e, se você estiver no UNIX normalmente não terá estas funções disponíveis.

Protótipos:

```
int getch (void);
int getche (void);
```

getch() espera que o usuário digite uma tecla e retorna este caractere. Você pode estar estranhando o fato de **getch()** retornar um inteiro, mas não há problema pois este inteiro é tal que quando igualado a um **char** a conversão é feita corretamente. A função **getche()** funciona exatamente como **getch()**. A diferença é que **getche()** gera um "echo" na tela antes de retornar a tecla.

Se a tecla pressionada for um caractere especial estas funções retornam zero. Neste caso você deve usar as funções novamente para pegar o código da tecla extendida pressionada.

A função equivalente a **getche()** no mundo ANSI é o **getchar()**. O problema com getchar é que o caracter lido é colocado em uma área intermediária até que o usuário digite um <ENTER>, o que pode ser extremamente inconveniente em ambientes interativos.

- putchar

Protótipo:

```
int putchar (int c);
```

putchar() coloca o caractere c na tela. Este caractere é colocado na posição atual do cursor. Mais uma vez os tipos são inteiros, mas você não precisa se preocupar com este fato. O header file é *stdio.h*.

Lendo e Escrevendo Strings

- gets

Protótipo:

```
char *gets (char *s);
```

Pede ao usuário que entre uma string, que será armazenada na string **s**. O ponteiro que a função retorna é o próprio **s**. **gets** não é uma função segura. Por quê? Simplesmente porque com **gets** pode ocorrer um estouro da quantidade de posições que foi especificada na string . Veja o exemplo abaixo:

```
#include <stdio.h>
int main()
{
   char buffer[10];
   printf("Entre com o seu nome");
   gets(buffer);
   printf("O nome é: %s", buffer);
   return 0;
}
```

Se o usuário digitar como entrada:

Renato Cardoso Mesquita

ou seja, digitar um total de 23 caracteres: 24 posições (incluindo o '\0') serão utilizadas para armazenar a string. Como a string buffer[] só tem 10 caracteres, os 14 caracteres adicionais serão colocados na área de memória subsequente à ocupada por ela, escrevendo uma região de memória que não está reservada à string. Este efeito é conhecido como "estouro de buffer" e pode causar problemas imprevisíveis. Uma forma de se evitar este problema é usar a função fgets, conforme veremos posteriormente

- puts

Protótipo:

int puts (char *s);

puts() coloca a string **s** na tela.

AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia nomes pelo teclado e os imprima na tela. Use as funções puts e gets para a leitura e impressão na tela.

Entrada e Saída Formatada

As funções que resumem todas as funções de entrada e saída formatada no C são as funções **printf()** e **scanf()**. Um domínio destas funções é fundamental ao programador.

- printf

Protótipo:

int printf (char *str,...);

As reticências no protótipo da função indicam que esta função tem um número de argumentos variável. Este número está diretamente relacionado com a string de controle **str**, que deve ser fornecida como primeiro argumento. A string de controle tem dois componentes. O primeiro são caracteres a serem impressos na tela. O segundo são os comandos de formato. Como já vimos, os últimos determinam uma exibição de variáveis na saída. Os comandos de formato são precedidos de **%**. A cada comando de formato deve corresponder um argumento na função **printf()**. Se isto não ocorrer podem acontecer erros imprevisíveis no programa.

Abaixo apresentamos a tabela de códigos de formato:

Código	Formato
%c	Um caracter (char)
%d	Um número inteiro decimal (int)
%i	O mesmo que %d
%e	Número em notação científica com o "e"minúsculo
%E	Número em notação científica com o "e"maiúsculo
%f	Ponto flutuante decimal
%g	Escolhe automaticamente o melhor entre %f e %e
%G	Escolhe automaticamente o melhor entre %f e %E
%o	Número octal
%s	String
%u	Decimal "unsigned" (sem sinal)
%x	Hexadecimal com letras minúsculas
%X	Hexadecimal com letras maiúsculas
%%	Imprime um %

%p	Ponteiro

Vamos ver alguns exemplos:

Código	Imprime
printf ("Um %%%c %s",'c',"char");	Um %c char
printf ("%X %f %e",107,49.67,49.67);	6B 49.67 4.967e1
printf ("%d %o",10,10);	10 12

É possível também indicar o tamanho do campo, justificação e o número de casas decimais. Para isto usa-se códigos colocados entre o **%** e a letra que indica o tipo de formato.

Um inteiro indica o tamanho mínimo, em caracteres, que deve ser reservado para a saída. Se colocarmos então **%5d** estamos indicando que o campo terá cinco caracteres de comprimento *no mínimo*. Se o inteiro precisar de mais de cinco caracteres para ser exibido então o campo terá o comprimento necessário para exibi-lo. Se o comprimento do inteiro for menor que cinco então o campo terá cinco de comprimento e será preenchido com espaços em branco. Se se quiser um preenchimento com zeros pode-se colocar um zero antes do número. Temos então que **%05d** reservará cinco casas para o número e se este for menor então se fará o preenchimento com zeros.

O alinhamento padrão é à direita. Para se alinhar um número à esquerda usa-se um sinal - antes do número de casas. Então **%-5d** será o nosso inteiro com o número mínimo de cinco casas, só que justificado a esquerda.

Pode-se indicar o número de casas decimais de um número de ponto flutuante. Por exemplo, a notação **%10.4f** indica um ponto flutuante de comprimento total dez e com 4 casas decimais. Entretanto, esta mesma notação, quando aplicada a tipos como inteiros e strings indica o número mínimo e máximo de casas. Então **%5.8d** é um inteiro com comprimento mínimo de cinco e máximo de oito.

Vamos ver alguns exemplos:

Código	Imprime
printf ("%-5.2f",456.671);	| 456.67|
printf ("%5.2f",2.671);	| 2.67|
printf ("%-10s","Ola");	|Ola |

Nos exemplos o "pipe" (|) indica o início e o fim do campo mas não são escritos na tela.

- scanf

Protótipo:

```
int scanf (char *str,...);
```

A string de controle str determina, assim como com a função **printf()**, quantos parâmetros a função vai necessitar. Devemos sempre nos lembrar que a função **scanf()** deve receber ponteiros como parâmetros. Isto significa que as variáveis que não sejam por natureza ponteiros devem ser passadas precedidas do operador **&**. Os especificadores de formato de entrada são muito parecidos com os de **printf()**. Os caracteres de conversão *d, i, u* e *x* podem ser precedidos por *h* para indicarem que um apontador para *short* ao invés de *int* aparece na lista de argumento, ou pela letra *l* (letra ele) para indicar que que um apontador para *long*

aparece na lista de argumento. Semelhantemente, os caracteres de conversão *e, f* e *g* podem ser precedidos por *l* para indicarem que um apontador para *double* ao invés de *float* está na lista de argumento. Exemplos:

Código	Formato
%c	Um único caracter (char)
%d	Um número decimal (int)
%i	Um número inteiro
%hi	Um short int
%li	Um long int
%e	Um ponto flutuante
%f	Um ponto flutuante
%lf	Um double
%h	Inteiro curto
%o	Número octal
%s	String
%x	Número hexadecimal
%p	Ponteiro

- sprintf e sscanf

sprintf e sscanf são semelhantes a printf e scanf. Porém, ao invés de escreverem na saída padrão ou lerem da entrada padrão, escrevem ou leem em uma string. Os protótipos são:

```
int sprintf (char *destino, char *controle, ...);
int sscanf (char *destino, char *controle, ...);
```

Estas funções são muito utilizadas para fazer a conversão entre dados na forma numérica e sua representação na forma de strings. No programa abaixo, por exemplo, a variável i é "impressa" em string1. Além da representação de i como uma string, string1 também conterá "Valor de i=" .

```
#include <stdio.h>
int main()
{
   int i;
   char string1[20];
   printf( " Entre um valor inteiro: ");
   scanf("%d", &i);
   sprintf(string1,"Valor de i = %d", i);
   puts(string1);
   return 0;
}
```

Já no programa abaixo, foi utilizada a função sscanf para converter a informação armazenada em string1 em seu valor numérico:

```
#include <stdio.h>
int main()
{
   int i, j, k;
   char string1[]= "10 20 30";
   sscanf(string1, "%d %d %d", &i, &j, &k);
   printf("Valores lidos: %d, %d, %d", i, j, k);
```

```
    return 0;
}
```

AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia (via teclado) e apresente uma matriz 3X3 na tela. Utilize os novos códigos de formato aprendidos para que a matriz se apresente corretamente identada. Altere os tipos de dados da matriz (int, float, double) e verifique a formatação correta para a identação. Verifique também a leitura e impressão de números hexadecimais.