**FIT - Faculdade de Informática de Taquara**
**Curso de Sistemas de Informação**

# Apostila de Programação II
# - Linguagem de Programação C -

**“Material até a Prova G1”**

**Profa. Flávia Pereira de Carvalho**

**Março de 2008**

## Sumário

*Página*

# 1 Introdução

A Linguagem de Programação C foi primeiramente criada por Dennis Ritchie e Ken Thompson nos laboratórios da empresa Bell, em 1972. C foi baseada na Linguagem B de Thompson, que por sua vez era uma evolução da Linguagem BCPL (o que nos leva a concluir que uma próxima evolução desta linguagem gerasse a linguagem P ;-D ). Esta linguagem foi inicialmente concebida para ser utilizada no Sistema Operacional Unix.

A definição de C está contida em um livro escrito pelo próprio Dennis Ritchie, juntamente com Brian Kernighan cujo título é **C - A Linguagem de Programação padrão ANSI**[1] *(American National Standards Institute)*. É um livro quase obrigatório para quem estiver estudando a linguagem C. Uma fonte alternativa de aprendizado são os milhares de tutoriais, apostilas e livros disponíveis na Internet. É bom prestar muita atenção neste tipo de fonte, pois nem sempre o material é confiável. Mas aí vai uma dica: o livro **C / C++ e Orientação a Objetos em Ambiente Multiplataforma** foi escrito pelo Prof. Sergio Villas-Boas da UFRJ e é encontrado na Internet facilmente para download gratuito (no formato PDF).

C é uma linguagem de programação genérica que é utilizada para criação de programas diversos como processadores de texto, planilhas eletrônicas, sistemas operacionais, programas de comunicação, programas para automação industrial, gerenciadores de bancos de dados, programas para solução de problemas da Engenharia, Física, Química e outras Ciências etc. É bem provável que o Navegador Web que você usou para acessar este material tenha sido escrito em C ou C++.

C é freqüentemente chamada de linguagem de médio nível para computadores. Isso não significa que C seja menos poderosa, difícil de usar ou menos desenvolvida que uma linguagem de alto nível como Pascal, tampouco implica que C seja similar à linguagem assembly e seus problemas correlatos aos usuários. C é tratada como uma linguagem de médio nível porque combina elementos de linguagens de alto nível com a funcionalidade da linguagem assembly.

Como uma linguagem de médio nível, C permite a manipulação de bits, bytes e endereços – os elementos básicos com os quais o computador funciona. Um código escrito em C é muito portável. **Portabilidade** significa que é possível adaptar um software escrito para um tipo de computador a outro. Por exemplo, se você pode facilmente converter um programa escrito para DOS de tal forma a executar sob Windows, então esse programa é portável. Por haver compiladores C para quase todos os computadores, é possível pegar um código escrito para uma máquina, compilá-lo e rodá-lo em outra com pouca ou nenhuma modificação. Esta portabilidade economiza tempo e dinheiro. Os compiladores C também tendem a produzir um código-objeto muito compacto e rápido.

Embora o termo “linguagem estruturada em blocos” não seja rigorosamente aplicável a C, ela é normalmente referida simplesmente como linguagem estruturada. C tem muitas semelhanças com outras linguagens estruturadas, como Pascal, por exemplo. A razão pela qual C não é, tecnicamente, uma linguagem estruturada em blocos, é que as linguagens estruturadas em blocos permitem que procedimentos e funções sejam declarados dentro de procedimentos e funções. No entanto, como **C não permite a criação de funções dentro de funções**, não pode ser chamada formalmente de uma linguagem estruturada em blocos [SCH97].

1 Esse livro pode ser adquirido diretamente na Editora Campus. Na página da disciplina de Programação II, na seção Downloads, está disponível o link direcionado para a compra. E também tem a disposição dos alunos na nossa biblioteca.

# 2 Compiladores e Interpretadores

Os termos compiladores e interpretadores referem-se à maneira como um programa é executado. Existem dois métodos gerais pelos quais um programa pode ser executado. Em teoria, qualquer linguagem de programação pode ser compilada ou interpretada, mas algumas linguagens geralmente são executadas de uma maneira ou de outra. A maneira pela qual um programa é executado não é definida pela linguagem em que ele é escrito. Interpretadores e compiladores são simplesmente programas sofisticados que operam sobre o código-fonte do seu programa.

Um interpretador lê o código-fonte do seu programa uma linha por vez, executando a instrução específica contida nessa linha. Um compilador lê o programa inteiro e converte-o em um código-objeto, que é uma tradução do código-fonte do programa em uma forma que o computador possa executar diretamente.

Quando um interpretador é usado, deve estar presente toda vez que você executar o seu programa. O processo é lento e ocorre toda vez que o programa for executado. Um compilador, ao contrário, converte seu programa em um código-objeto que pode ser executado diretamente por seu computador. Como o compilador traduz seu programa de uma só vez, tudo o que você precisa fazer é executar se programa diretamente, geralmente apenas digitando seu nome. Assim, o tempo de compilação só é gasto uma vez, enquanto o código interpretado incorre neste trabalho adicional cada vez que o programa executa [SCH97].

## 2.1 Compiladores C

Existem diversos compiladores C / C++ disponíveis para os programadores. A decisão sobre qual compilador utilizar pode ser baseada em vários fatores, como por exemplo:

- Qualidade do compilador

  - é rápido

  - está conforme com a padronização da linguagem

  - a interface com o usuário é agradável - possui ou não um IDE *(Integrated Development Enviroment)*

  - possui diversas opções de compilação; etc.

- Sistema(s) Operacional(is) que o compilador funciona (e gera códigos)
- Custo do compilador
- Documentação disponível e Suporte

Dentro deste cenário, temos também um fator determinante: quais são os compiladores mais utilizados atualmente? Esta informação é muito importante, pois é indicado que estejamos habituados e com experiência naquele compilador que provavelmente tenhamos que utilizar em algum futuro emprego ou projeto de pesquisa que venhamos a nos engajar.

### 2.1.1 Compilador GCC (GNU Compiler Collection)

O compilador GCC *(http://www.gnu.org/software/gcc/gcc.html)* é da GNU[2] *(Free Software Foundation).* É sem dúvida um "líder" no mercado de compiladores C atualmente e é o que iremos utilizar na disciplina de Programação II. É o compilador padrão do sistema operacional GNU/Linux e também foi adotado por (ou portado para) vários outros sistemas operacionais (inclusive S.O. comerciais pagos), tais como HP-UX, MS-Windows, MS-DOS, IBM OS2, IBM AIX, SUN OS, SUN Solaris etc, além é claro do próprio GNU/Linux, onde este compilador é utilizado no desenvolvimento do próprio sistema operacional e de todas as ferramentas nele disponíveis (ou seja, é o compilador 'nativo' do Linux).

Outro detalhe importante é que todas as distribuições Linux já vêm com um compilador GCC disponível, ou seja, se você tem um computador com Linux instalado, você tem um compilador GCC pronto para ser utilizado. Mas caso você precise instalar o GCC em outro S.O., tal como o Windows, vá em: *http://gcc.gnu.org/install/binaries.html* (embora em um primeiro momento, por motivos de facilidade na utilização, seja extremamente recomendado que você utilize o GCC no Linux, e só após estar familiarizado tente instalá-lo em algum outro S.O.).

### 2.1.2 Compilador Dev-C++

Compilador *Free Software* (GPL[3] - *General Public License*), para Windows, com um IDE 'respeitável' (editor, compilador e debug integrados). Para efetuar o download gratuito, acesse:

*http://fit.faccat.br/~fpereira/devcpp-4.9.9.2_setup.zip*

Este compilador é o mais recomendado para quem quiser trabalhar no Windows.

### 2.1.3 Compilador C++ Builder - Borland

Outro compilador C muito conhecido e utilizado é o da empresa Borland[4], certamente uma das empresas mais conhecidas na área de linguagens de programação (Delphi, Turbo Pascal, Turbo C, Kylix etc.).

O compilador C++ Builder pode ser baixado gratuitamente (versão de avaliação - 60 dias) no endereço: *http://www.borland.com/bcppbuilder/tryitnow.html*.

### 2.1.4 Compilador Turbo C Borland

Este é um dos compiladores C mais conhecidos (e antigos) na micro-informática. A Borland disponibiliza o Turbo C gratuitamente no site da empresa (após um rápido cadastro). Você pode baixá-lo diretamente em: *http://fit.faccat.br/~fpereira/tc201.zip* Este compilador roda em "modo MS-DOS". É muito simples, mas funciona perfeitamente em muitas versões de Windows.

Dicas para instalação: ao descompactar o arquivo, serão criadas três pastas (disk1, disk2 e disk3). A forma de instalação correta solicita que cada uma destas pastas seja descompactada em um disquete de 3,5" e a instalação seria então iniciada a partir do Disk1. Mas também é possível realizar o procedimento de instalação da seguinte forma:

1. Descompactar todos os arquivos em uma única pasta
2. Iniciar o processo de instalação executando o arquivo 'install.exe'

2 Maiores informações sobre o projeto GNU em: *http://www.gnu.org/*
3 Maiores informações em: *http://www.gnu.org/copyleft/gpl.html*
4 Maiores informações sobre a empresa Borland em: *http://www.borland.com/br/*

3. Tão logo inicia a instalação, o software instalador solicita a informação sobre o **drive de origem da instalação**: digite o drive onde a pasta foi criada (possivelmente C:) e confirme o caminho da pasta que o próprio instalador irá sugerir.

Tudo pronto: vá em File - New e digite seu primeiro programa. Para compilá-lo e rodá-lo vá em Run - Run.

### 2.1.5 Compilador Visual C# - Microsoft

O compilador C da Microsoft é outro com muitos usuários. Algumas informações em:

*http://www.microsoft.com/brasil/msdn/csharp/default.mspx*

*http://www.microsoft.com/brasil/msdn/csharp/articles.mspx.*

## 3 Considerações Iniciais e Dicas

Neste capítulo serão apresentadas algumas dicas importantes sobre a linguagem C e alguns comentários iniciais para que possamos iniciar os primeiros programas.

Antes disso é necessário dizer algumas palavras sobre C++. Algumas vezes os novatos confundem o que é C++ e como difere de C. C++ é uma versão estendida e melhorada de C que é projetada para suportar programação orientada a objetos (OOP, do inglês *Object Oriented Programming*). C++ contém e suporta toda linguagem C e mais um conjunto de extensões orientadas a objetos. Ou seja, C++ é um superconjunto de C. Como C++ é construída sobre os fundamentos de C, você não pode programar em C++ se não entender C.

Hoje em dia, e por muitos anos ainda, a maioria dos programadores ainda escreverá, manterá e utilizará programas C, e não C++. Como mencionado, C suporta programação estruturada, que tem se mostrado eficaz ao longo de todos estes anos nos quais tem sido largamente usada. C++ é projetada principalmente para suportar OOP, que incorpora os princípios da programação estruturada, mas inclui objetos. Embora a OOP seja muito eficaz para uma certa classe de tarefas de programação, muitos programas não se beneficiam da sua aplicação.

Com relação aos compiladores: todos os compiladores que podem compilar programas C++ também podem compilar programas C.

### 3.1 A Linguagem C é “case sensitive”

Um ponto de suma importância: C é *"case sensitive"*, ou seja, maiúsculas e minúsculas fazem diferença. Se declararmos uma variável com o nome ‘soma’ ela será diferente de Soma, SOMA, SoMa ou sOmA. Da mesma maneira, os **comandos** (todas as palavras-chave) da linguagem 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.

### 3.2 Variáveis

Primeiramente com relação aos nomes das variáveis (e também das funções) -- em C os identificadores podem ter qualquer nome se duas condições forem satisfeitas:

- O nome deve começar com uma letra ou sublinhado ( _ )
- Os caracteres subseqüentes devem ser letras, números ou sublinhado ( _ ).

Além dessas duas condições acima, ainda há mais duas restrições:

- O nome de uma variável não pode ser igual a uma palavra reservada (palavra-chave), nem igual ao nome de uma função declarada pelo programador, ou pelas bibliotecas do C. Variáveis de até 32 caracteres são aceitas.
- É bom sempre lembrar que C é uma linguagem *"case sensitive"* e, portanto deve-se prestar atenção às maiúsculas e minúsculas.

### 3.2.1 Declarações

Todas as variáveis devem ser declaradas antes de serem usadas. Uma declaração especifica um tipo, e é seguida por uma lista de uma ou mais variáveis daquele tipo, tal como nos seguintes exemplos:

```
int inicio, fim, numero;
float valor;
short int codigo;
char c, linha [100];
```

### 3.2.2 Notação Húngara

Programadores normalmente não escolhem os nomes de variáveis, funções e constantes ao mero acaso. Normalmente elegem nomes que signifiquem e indiquem o motivo da existência daquela variável ou função dentro do programa. O problema é que determinada palavra ou sigla pode significar muito para uma determinada pessoa e nada para outra. Por este motivo é muito interessante que exista uma convenção quanto a nomenclatura para esses identificadores.

Você já deve ter ouvido falar sobre algo chamado de “notação húngara”. Talvez nunca tenha ouvido, mas provavelmente já viu. É quando você tem variáveis parecidas com as seguintes:

```
intComputadores = 20
dtHoje = #07/01/05#
strNomeComputador = "server1"
```

A nomeação de variáveis dessa forma é uma prática comum de programação. Tudo o que você está fazendo é criando um nome de variável, como `intComputadores` e, em seguida, prefixando-a com o tipo de informação que será armazenado nela. Portanto, se você estiver usando a variável `intComputadores` para manter o número de computadores, usará um inteiro para designar o número. Portanto, o prefixo `int`. Isso é tudo. Isso ajuda a evitar que você pense que talvez `intComputadores` mantenha os nomes dos computadores, o que seria uma seqüência de caracteres e não um inteiro.

**Por que húngara?** Porque a primeira pessoa a propor esse tipo de nomenclatura de variáveis foi o ex-engenheiro da Microsoft, Charles Simonyi, que era húngaro.

Mais informações em:

*http://www.microsoft.com/technet/scriptcenter/guide/sas_sbp_xmzd.mspx?mfr=true*

*http://msdn2.microsoft.com/en-us/library/aa260976.aspx*

A notação húngara *(Hungarian Notation)* é uma das convenções para nomenclatura mais difundidas e utilizadas. Normalmente, um código sofisticado e que tenha pretensões de ser divulgado ou publicado largamente, necessita obedecer esta notação para ser compreendido e aceito pela comunidade científica e comercial. Outro pequeno exemplo pode ser visto a seguir:

```
string sPrimeiroNome; //s é o prefixo para o tipo String
char cLetra; //c é o prefixo para o tipo Char
button butOk; //but é o prefixo para o tipo Button
```

Por estes motivos, grandes empresas e *softwares houses* costumam exigir de seus programadores e fornecedores de software a utilização desta convenção. Xerox, Apple, 3Com e Microsoft são exemplos de empresas que utilizam esta convenção como padrão de nomenclatura para seus identificadores de programas.

### 3.2.3 Inicialização de Variáveis (Atribuição)

Você pode dar à maioria das variáveis em C um valor, no mesmo momento em que elas são declaradas, ou seja, pode inicializar a variável juntamente com sua declaração, colocando um **sinal de igual** ( = ) e uma constante após o nome da variável. A forma geral de uma inicialização é:

```
tipo nome_da_variável = constante;
```

➔ Alguns exemplos de inicializar juntamente ao declarar são:

```
char letra = ‘a’;
int valor = 0;
float preço = 123.23;
```

A linguagem C permite que você atribua o mesmo valor a muitas variáveis usando atribuições múltiplas em um único comando. Por exemplo, esse fragmento de programa abaixo, atribui a **x, y e z** o valor **0**:

```
x = y = z = 0;
```

Em programas profissionais, valores comuns são atribuídos a variáveis usando esse método.

O qualificador `const` pode ser aplicado à declaração de qualquer variável para especificar que seu valor não deve ser mudado:

```
const float salario_minimo = 230.00;
const char mensagem[] = "Ola";
```

## 3.3 Cartão de Referência da Linguagem ANSI C

Cartões de Referência são clássicos entre programadores de praticamente to das as linguagens. Nestes cartões, apesar de seu tamanho reduzido (uma folha impressa nos dois lados e dobrada em forma de folder), normalmente é possível consultar todos os tipos de dados da linguagem, formas e exemplos de declaração, tipos de estruturas de decisão e repetição, as sintaxes para cada uma destas definições etc. São cartões de referência completos e muito úteis no dia-a-dia.

Um cartão de referência da ANSI C muito bom, criado por Joseph Silverman, da Brown University e liberado para cópias gratuitas pode ser encontrado para download em:

*http://fit.faccat.br/~fpereira/C_reference_Card_ANSI.pdf*

**Observação:** Não esqueça que este cartão pode (deve) ser impresso frente-e-verso para ficar prático!

## 3.4 Tipos de Dados

- **Tipos Básicos:** os dados podem assumir cinco tipos básicos em C que são:

`char:` Caractere - o valor armazenado é um caractere. Caracteres geralmente são armazenados em códigos (usualmente o código ASCII).

`int:` Número inteiro é o tipo padrão e o tamanho do conjunto que pode ser representado normalmente depende da máquina em que o programa está rodando.

`float:` Número em ponto flutuante de precisão simples. São conhecidos normalmente como números reais.

`double:` Número em ponto flutuante de precisão dupla.

`void:` Este tipo serve para indicar que um resultado não tem um tipo definido. Uma das aplicações deste tipo em C é criar um tipo vazio que pode posteriormente ser modificado para um dos tipos anteriores. O tipo **void** declara explicitamente uma função que não retorna valor algum ou cria ponteiros genéricos, como será visto mais adiante.

- **Modificadores dos Tipos Básicos:** modificadores podem ser aplicados a estes tipos. Estes modificadores são palavras que alteram o tamanho do conjunto de valores que o tipo pode representar. Por exemplo, um modificador permite que possam ser armazenados números inteiros maiores. Um outro modificador obriga que só números sem sinal possam ser armazenados pela variável. Deste modo não é necessário guardar o bit de sinal do número e somente números positivos são armazenados. O resultado prático é que o conjunto praticamente dobra de tamanho. A Tabela abaixo mostra todos os tipos básicos definidos no padrão ANSI.

| Tipo | Tamanho em Bytes | Faixa Mínima |
|---|---|---|
| **Char** | 1 | -127 a 127 |
| **unsigned char** | 1 | 0 a 255 |
| **signed char** | 1 | -127 a 127 |
| **Int** | 4 | -2.147.483.648 a 2.147.483.647 |
| **unsigned int** | 4 | 0 a 4.294.967.295 |
| **signed int** | 4 | -2.147.483.648 a 2.147.483.647 |
| **short int** | 2 | -32.768 a 32.767 |
| **unsigned short int** | 2 | 0 a 65.535 |
| **Signed short int** | 2 | -32.768 a 32.767 |
| **long int** | 4 | -2.147.483.648 a 2.147.483.647 |
| **Signed long int** | 4 | -2.147.483.648 a 2.147.483.647 |
| **unsigned long int** | 4 | 0 a 4.294.967.295 |
| **Float** | 4 | Seis digitos de precisão |
| **Double** | 8 | Dez digitos de precisão |
| **long double** | 10 | Dez digitos de precisão |

**Tipos básicos definidos no padrão ANSI**

Percebemos então que há uma série de qualificadores que podem ser aplicados aos tipos básicos: `short` e `long`.

```
short int iTesteShort;
long int iTesteLong;
```

A intenção é que `short` e `long` devam prover tamanhos diferentes de inteiros onde isso for possível; `int` normalmente será o tamanho natural para uma determinada máquina. short ocupa normalmente 16 bits, long 32 bits, e int, 16 ou 32 bits (de acordo com a máquina). Cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que short e int devem ocupar pelo menos 16 bits, long pelo menos 32 bits, e short não pode ser maior que int, que não é maior do que long.

Existem também os qualificadores `signed` e `unsigned`, que podem ser aplicados a `char` ou a qualquer inteiro. Números unsigned são sempre positivos ou 0.

O tipo `long double` especifica ponto flutuante com precisão estendida. Assim como os inteiros, os tamanhos de objetos em pontos flutuantes são definidos pela implementação.

# 4 Início do Aprendizado: dois primeiros programas

Nas próximas seções deste capítulo são apresentados exemplos iniciais de programas feitos na linguagem C, explicando detalhadamente cada linha do código.

## 4.1 Primeiro exemplo de um programa em C

```
1   #include <stdio.h>
2   /*Um Primeiro Programa*/
3   main ()
4   {
5      printf ("Ola! Eu estou vivo!\n"); //Escreve na tela
6   }
```

Compilando e executando este programa, ele escreve na tela: **Ola! Eu estou vivo!**

➔ **Analisando o programa acima por partes** (linhas numeradas para facilitar)**:**

**1)** `#include <stdio.h>` diz ao compilador que ele deve incluir o arquivo-cabeçalho padrão da linguagem C **stdio.h**. Nesse arquivo existem declarações de funções úteis para entrada e saída de dados onde:

std = standard, que significa padrão, em inglês

io = Input/Output, entrada e saída

**Então:** stdio = entrada e saída padronizadas.

Toda vez que você quiser usar uma destas funções deve incluir este comando. A linguagem C possui diversos arquivos-cabeçalhos.

**2)** Quando fazemos um programa, uma boa idéia é usar **comentários** que ajudem a elucidar o funcionamento do mesmo. No exemplo apresentado temos um comentário:

`/*Um Primeiro Programa*/`

O compilador desconsidera qualquer coisa que esteja começando com `/*` e terminando com `*/`. Um comentário pode, inclusive, ter mais de uma linha. Aliás, se o comentário for de apenas uma linha, pode-se usar `//`.

**3)** `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. No exemplo, a função main não recebe argumentos (os parênteses estão vazios).

**4)** O corpo (conteúdo) da função é delimitado por chaves `{ }`. O código que estiver dentro das chaves será executado seqüencialmente quando a função for chamada.

**5)** 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 a seguir) `"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 `;` e que strings são delimitadas por aspas duplas.

**6)** As chaves, ou seja, o terminador `}` indica o fim da área da função.

Programas em C são definidos em arquivos texto que normalmente possuem a **extensão .c**.

## 4.2 Segundo exemplo de um programa em C

```
#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);  //Lê do teclado
   Anos=Dias/365.25;  //Conversao Dias em Anos
   printf ("\n\n %d dias equivalem a %f anos.\n",Dias,Anos);
   /* experimente trocar o %f por %5.2f */
   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 - números com ponto). As variáveis declaradas como ponto flutuante existem para armazenar números que possuem casas decimais, como por exemplo, 5.1497. A palavra **int** antes de **main** 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.

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 do teclado. 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, em C, quando temos mais de um parâmetro para uma função, eles são separados por vírgula. Temos então uma expressão matemática simples que atribui a `Anos` o valor de `Dias` dividido por 365.25.

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.

## 5 Auto Avaliação: 1ª. Parte da Disciplina

➔ 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);
}
```

## 6 Matrizes Unidimensionais - Vetores (ou Array)

Uma matriz é uma coleção de variáveis do mesmo tipo que é referenciada por um nome comum. Um elemento específico em uma matriz é acessado por meio de um **índice**. Os índices **iniciam em 0** em C.

A matriz mais comum em C é a de string, que é simplesmente uma matriz de caracteres terminada por um nulo. A forma geral para declarar uma matriz unidimensional é:

```
tipo nome_vetor [tamanho];
```

**Exemplos:**

**1)**
```
int codigos[100];
```

**2)**
```
char nomes[80];   //isto eh uma string em C
```

**3)**
```
float salarios[30];
```

No exemplo número **1** apresentado acima, estamos declarando um vetor de inteiros com 100 posições, chamado `codigos`. Os índices vão de **codigos[0]** até **codigos[99]**.

No exemplo **2**, temos um vetor de caracteres com 80 posições, de **nome[0]** até **nome[79]**. Esta é a forma básica para declarações de strings em C.

No exemplo **3**, temos um vetor float com 30 posições (índices de 0 a 29), chamado `salarios`.

Por exemplo, o seguinte programa carrega uma matriz inteira com os números de 0 a 99:

```
void main (void)
{
   int x[100];
   int t;
   for (t=0; t<100; ++t)
   {
      x[t]=t;
   }
}
```

## 7 Funções: scanf( ) e printf( )

As funções scanf( ) e printf( ) são respectivamente funções de entrada e saída de dados, ou seja, scanf( ) é uma função de leitura das informações (do teclado) e printf ( ) é uma função de escrita das informações.

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

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

Teremos, na **string de controle**, uma descrição de tudo que a função vai colocar na tela. A string de controle mostra não apenas os caracteres que devem ser colocados na tela, mas também quais as variáveis e suas respectivas posições. Isto é feito usando-se os códigos de controle, que usam a notação **%**. Na string de controle indicamos quais, de qual tipo e em que posições estão as variáveis a serem apresentadas. É muito importante que, para cada código de controle, tenhamos um argumento na lista de argumentos. Resumindo, a função printf ( ) converte, formata e imprime seus argumentos na saída padrão (normalmente a tela) sob o controle da string_de_controle.

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

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

A função scanf ( ) lê caracteres da entrada padrão (normalmente o teclado), interpretando-os de acordo com o formato especificado na string_de_controle, e armazena os resultados nos argumentos descritos.

Na tabela a seguir são apresentados alguns dos **códigos %:**

| Código | Significado |
|---|---|
| **%d** | Inteiro Decimal |
| **%f** | Float: número em ponto flutuante (número real, com ponto – casas decimais) |
| **%c** | Caractere |
| **%s** | String: série de caracteres |
| **%o** | Inteiro Octal |
| **%x** | Número Hexadecimal |
| **%u** | Decimal sem Sinal |
| **%l** | Inteiro Longo |
| **%%** | Coloca na tela um **%** |

**Códigos de Formatação das Funções: scanf( ) e printf( )**

Além dos códigos de controle, existem também os códigos especiais, tais como o '**\n**' que já vimos anteriormente. Na tabela abaixo estão listados esses **códigos especiais**:

| Código | Significado |
|---|---|
| \n | Nova Linha |
| \t | Tab |
| \b | Retrocesso |
| \" | Aspas |
| \\ | Contrabarra |
| \f | Salta página de Formulário |
| \0 | Nulo |

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 eh um exemplo","Este") ➔ Este eh um exemplo
printf ("%s%d%%","Juros de ",10) ➔ Juros de 10%
```

➔ Veja no exemplo abaixo as diversas possibilidades do uso dos códigos de formatação nas funções scanf( ) e printf( ):

```
main()
{
   char a;
   printf ("Digite um caractere e veja-o em decimal, octal e
            hexadecimal: ");
   scanf ("%c",&a);
   printf ("\n %c = %d em decimal, %o em octal e %x em
            hexadecimal.\n",a,a,a,a);
}
```

➔ Um exemplo de resultado de execução do programa acima:

Digite um caractere e veja-o em decimal, octal e hexadecimal:  m
m = 109 em decimal,  155 em octal  e  6D em hexadecimal.

## 8 Caracteres: Lendo e Escrevendo

Os caracteres são um tipo de dado: o **char**. A linguagem C trata os caracteres ('a', 'b', 'x' etc.) como sendo variáveis de 1 byte (8 bits). Um bit é a menor unidade de armazenamento de informações em um computador. Os inteiros (int) têm um número maior de bytes. Dependendo da implementação do compilador, eles podem ter 2 bytes (16 bits) ou 4 bytes (32 bits). Em 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 cLetra;
    cLetra='D';
    printf ("%c",cLetra);
    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 cLetra;
   cLetra='D';
   printf ("%d",cLetra); /*Imprime o caractere 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 a função mais usada é `getchar()`. Esta função **retorna o caractere pressionado**. O equivalente a `getchar()` para impressão é a função `putchar()`. Eis um exemplo que usa as funções **getchar( )** e **putchar( )**:

```
#include <stdio.h>
int main ()
{
   char Ch;
   Ch=getchar();
   putchar(Ch);
   printf ("\nVoce pressionou a tecla %c\n",Ch);
   return(0);
}
```

Programa equivalente, **sem usar getchar( ):**

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

## 9 Introdução a alguns Comandos de Controle de Fluxo

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

### 9.1 Comando if

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

```
if (condição) comando;
else comando;
```

A condição do comando **if** é uma expressão que será avaliada. Abaixo está apresentado 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 agrupá-los em um bloco (com chaves) 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 valor de **num** fosse modificado e fazendo com que a declaração fosse executada sempre. Este problema gera erros freqüentes entre iniciantes, portanto, muita atenção deve ser tomada.

➔ Mais um exemplo (utilizando **if-else**):

```
#include <stdio.h>
int main()
{
    int iNum = 5;
    if (iNum > 10)
    {
        printf ("maior que 10\n");
    }
    else
    {
        printf ("menor que 10\n");
    }
    return 0;
}
```

➔ Outra forma de escrever a mesma estrutura **if-else** do exemplo acima, seria:

```
if (iNum > 10) printf ("maior que 10\n"); else printf ("menor que
10\n");
```

### 9.1.1 Ifs Aninhados

```
if (i)
{
    if (j) comando 1;
    if (k) comando 2;
    else comando 3;  //este else esta associado ao if(k)
}
else comando 4;       //este else esta associado ao if(i)
```

➔ Uma seqüência de **ifs** aninhados:

```
if (expressao) comando;
else
    if (expressao) comando;
    else
        if(expressao) comando;
```

➔ ou, para diminuirmos a 'altura' do código fonte:

```
if (expressao) comando;
else  if (expressao) comando;
    else  if(expressao) comando;
```

# 10 Operadores

Neste capítulo são apresentadas tabelas com os operadores da linguagem C: aritméticos, relacionais e lógicos.

## 10.1 Operadores Aritméticos

| Operador | Significado |
|---|---|
| **-** | Subtração |
| **+** | Adição |
| ***** | Multiplicação |
| **/** | Divisão |
| **%** | Módulo da divisão (resto) |
| **--** | Decremento |
| **++** | Incremento |

**Operadores Aritméticos**

A operação de somar ou subtrair 1 de um contador é muito comum. Os operadores unários `++` e `--` fazem isso diretamente sem a necessidade do comando de atribuição:

```
i = i + 1;  é equivalente a  i++;
i = i - 1;  é equivalente a  i--;
```

Podem ser usados à esquerda (prefixo) ou à direita (posfixo), sendo que:

- **à esquerda:** incrementa/decrementa e a seguir usa o valor.
- **à direita:** usa o valor e a seguir incrementa/decrementa.

```
i++;   ou   ++i;
i--;   ou   --i;
```

Em ambos os casos acima, a variável `i` é incrementada ou decrementada. Porém a expressão `++i` incrementa `i` *antes* de usar seu valor, enquanto `i++` incrementa `i` *após* seu valor ser usado. Isso significa que num contexto onde o valor é usado, `++i` e `i++` são diferentes. Por exemplo: se `i` é 5, então

```
x = i++;
```

Atribui **5** a `x`, mas

```
x = ++i;
```

Atribui **6** a `x`. Em ambos os casos, `i` torna-se **6**. Num contexto onde o valor não é usado e queremos apenas o efeito de incrementar ou decrementar, a escolha de prefixação ou posfixação é de acordo com o gosto do programador.

**Exemplos:**

`b = ++a;` equivalente a **{a = a + 1; b = a;}**

`b = a++;` equivalente a **{b = a; a = a + 1;}**

`b = --a;` equivalente a **{a = a – 1; b = a;}**

`b = a--;` equivalente a **{b = a; a = a - 1;}**

Esses operadores quando usados no meio de comandos, confundem um pouco. É conveniente usá-los em sua forma mais simples.

**Exemplos:**

```
a = 5;  b = a*(a++);
```

**b** sai valendo 5*5=25 e **a** sai valendo 6

```
a = 5;  b = a*++a;
```

**b** sai valendo 5*6=36 e **a** sai valendo 6

```
a = 5;  b = a*a++;
```

**b** sai valendo 5*5=25 e **a** sai valendo 6

## 10.2 Operadores Relacionais

| Operador | Significado |
|---|---|
| == | igual |
| != | diferente de *(not equal)* |
| > | maior que |
| < | menor que |
| >= | maior ou igual |
| <= | menor ou igual |

**Operadores Relacionais**

## 10.3 Operadores Lógicos

| Operador | Ação |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |

**Operadores Lógicos**

A idéia de verdadeiro e falso é a base dos conceitos dos operadores lógicos e relacionais. Em C, verdadeiro é qualquer valor diferente de zero e falso é zero. As expressões que usam operadores relacionais ou lógicos devolvem **0** para falso e **1** para verdadeiro. A tabela verdade dos operadores lógicos é apresentada a seguir, usando 0s e 1s:

| p | q | p&&q | p\|\|q | !p |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |

**Tabela Verdade**

Na linguagem C é permitido combinar diversas operações em uma expressão como mostrado abaixo:

```
10>5 && !(10<9) || 3<=4
```

➔ Qual é o resultado da expressão acima: verdadeiro ou falso?

## 10.4 Operador de Endereço &

A memória do computador é dividida em bytes, e estes bytes são numerados de 0 até o limite de memória do computador em questão. Estes números são chamados de 'endereços' de bytes. Um endereço é o nome que o computador usa para identificar a variável.

Toda variável ocupa uma certa localização na memória, e seu endereço é o do primeiro byte ocupado por ela. Um inteiro ocupa 2 bytes. Se declararmos uma variável `n` como inteira e a ela atribuirmos o valor 2, quando `n` for referenciada devolverá 2. Entretanto, se referenciarmos `n` precedido de & **(&n)**, teremos como retorno o endereço do primeiro byte onde `n` está armazenado na memória do computador.

O programa a seguir imprime o **conteúdo** da variável `n` e o **endereço físico** de `n` na memória:

```
main()
{
   int iNum;
   iNum=2;
   printf ("Valor = %d, Endereço = %u", num, &num);
}
```

Um endereço na memória é visto como um número inteiro sem sinal, por isso usamos o código de formatação `%u`.

A saída deste programa varia conforme a máquina e a memória do equipamento (e o momento em que for utilizado, mesmo estando na mesma máquina). Mas um exemplo de saída é:

**Valor = 2, Endereço = 1370**

## 10.5 Prioridades dos Operadores

A precedência dos operadores aritméticos é a seguinte:

| | |
|---|---|
| **Mais alta** | **++ --** |
| | **-** |
| | *** / %** |
| **Mais baixa** | **+ -** |

**Prioridades dos Operadores Aritméticos**

A tabela seguinte mostra a precedência dos operadores relacionais e lógicos:

| | |
|---|---|
| **Mais alta** | **!** |
| | **> >= < <=** |
| | **== !=** |
| | **&&** |
| **Mais baixa** | **\|\|** |

**Prioridades dos Operadores Relacionais e Lógicos**

Como no caso das expressões aritméticas, é possível usar parênteses para alterar a ordem natural de avaliação de uma expressão relacional e/ou lógica. Por exemplo,

```
!0 && 0 || 0
```

é falso. Porém, quando adicionamos parênteses, como abaixo, o resultado é verdadeiro:

```
!(0 && 0) || 0
```

Lembre-se de que toda expressão relacional e lógica produz como resultado 0 ou 1. Então, o seguinte fragmento de programa não apenas está correto, como imprimirá o número 1 na tela:

```
int x;
x = 100;
printf (“%d”, x>10);
```

# 11 Estruturas de Repetição (Laços)

Basicamente, existem três laços em C (assim como em todas as linguagens de programação modernas), sendo eles:

- Um laço com teste na parte "superior" (teste de entrada no laço)
- Outro laço com o teste na parte "inferior" (teste de saída do laço)
- E por último, um laço adequado para situações em que sabemos o número pré-determinado de repetições que deverão ser realizadas.

## 11.1 While (Enquanto-Faça)

O laço **while** é o que realiza o **teste na entrada** da estrutura, ou seja, o(s) comando(s) internos ao **while** só serão realizados se o teste for verdadeiro e enquanto o teste for verdadeiro. Sua sintaxe é:

```
while (condição) //é necessário usar o delimitador de blocos { } caso
                    haja mais de um comando interno ao while
   comando;
```

➔ Exemplo de utilização do laço **while**:

```
#include <stdio.h>
void main (void)
{
  int iNum;
  printf ("Por favor digite um numero (diferente de zero): ");
  scanf ("%d",&iNum);

  while (iNum==0)
  {
      printf ("Voce Digitou um numero invalido (zero). Por favor
                  digite novo numero: ");
      scanf ("%d",&iNum);
  }
  printf ("OK! Voce digitou um numero valido... obrigado!");
}
```

## 11.2 Do-While (Faça-Enquanto)

O laço **do-while** é o que realiza o **teste ao final da estrutura**, ou seja, o(s) comando(s) interno(s) ao laço serão executados **no mínimo uma vez**, certamente. Ao final da primeira execução é que o teste será realizado e então a seqüência de repetições poderá (ou não) ser interrompida. Sua sintaxe geral é:

```
do {
comando; // os delimitadores { } só são obrigatórios caso exista mais
            de um comando dentro da estrutura.
} while (condição);
```

➔ Exemplo de utilização do laço **do-while**:

```
#include <stdio.h>
void main (void)
{
  int iNum;
  do{
        printf ("Por favor digite um numero (diferente de zero): ");
        scanf ("%d",&iNum);
  }while (iNum == 0);
  printf ("OK! Voce digitou um numero valido... obrigado!");
}
```

## 11.3 For (Para)

O *loop* (laço) **for** é usado (normalmente) para repetir um comando, ou bloco de comandos**, um número pré-determinado de vezes**. Se não soubermos o número pré-determinado de repetições que serão necessárias, é provável que as estruturas **while** ou **do-while** sejam mais adequadas. Mas nada impede que 'truques' de programação sejam criados para que o laço **for** seja utilizado em situações diversas. Sua forma geral é:

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

A declaração no comando **for** também pode ser um bloco **{ }**.

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 palavra[100];  /*String, ate 99 caracteres*/
  int i, cont;
  printf("\n\n Digite uma palavra: ");
  scanf ("%s", palavra);  /*Le a string*/
  printf("\n\n Palavra digitada:\n %s", palavra);
  cont = 0;
  for (i=0; palavra[i] != '\0'; i=i+1)
  {
  if ( palavra[i] == 'c' )  /*Se for a letra 'c'*/
  cont = cont+1;  /*Incrementa o contador de caracteres*/
  }
  printf("\n Numero de caracteres c = %d", cont);
  return(0);
}
```

Note o teste que está sendo feito no **for**: o caractere armazenado em `palavra[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 maiusculas*/
#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 seqüencial.

**<u>AVALIAÇÃO</u> - Veja como você está:**

**a)** Explique porque está errado fazer

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

→ O que irá acontecer?

### 11.4 Exercícios de Fixação

**1)** Escreva um programa que armazene, em um vetor, os valores de 0 a 100. Após, todos os valores deverão ser escritos (um ao lado do outro, com um espaço separando cada número).

**1.1)** O mesmo exercício anterior, mas agora os números deverão ser escritos um abaixo do outro.

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

**3)** Escreva um programa que leia 10 valores do teclado. Imprima na tela a média dos 10 valores lidos.

**4)** Escreva um programa que leia 10 caracteres do teclado. Após, todos os caracteres deverão ser escritos na tela, na mesma ordem em que foram digitados.

**4.1)** O mesmo exercício anterior, mas agora os caracteres devem ser escritos na tela na ordem INVERSA a que foram digitados.

## 12 Strings

Na linguagem C **uma string é um vetor de caracteres terminado com um caractere nulo**. O caractere 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'`. Veremos a seguir os fundamentos necessários para que possamos utilizar 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 **"casa"** nela. Teremos:

| c | a | s | a | \0 | | |
|---|---|---|---|---|---|---|

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

**Atenção:** Existe um detalhe muito importante quanto a leitura de strings pelo teclado. O identificador (nome) de um vetor/matriz retorna automaticamente o endereço inicial na memória que este vetor está armazenado. Assim sendo, é **errado utilizar o operador de endereço (&) junto ao nome do vetor**. Por exemplo, quando fazemos referência ao vetor **palavra** do código abaixo, é equivalente a **&palavra[0]**.

```
#include <stdio.h>
int main ()  /*Este código possui um ERRO!!! O & no scanf do vetor/string*/
{
   char sPalavra[100];
   printf ("Digite uma palavra: ");
   scanf ("%s", &sPalavra);
   printf ("\n\n Voce digitou %s",sPalavra);
   return(0);
}
```

➔ Para que este mesmo código fique correto, basta **retirar o operador (&)** que está junto ao nome da string, ficando então da seguinte forma:

```
#include <stdio.h>
int main ()   /*Este codigo ESTA CORRETO!! FOI RETIRADO O & no scanf do
                  vetor/string*/
{
   char sPalavra[100];
   printf ("Digite uma palavra: ");
   scanf ("%s", sPalavra);
   printf ("\n\n Voce digitou %s",sPalavra);
   return(0);
}
```

➔ Neste programa acima, o tamanho máximo da string que você pode entrar é uma string de 99 caracteres.

## 12.1 Lendo Strings do Teclado: com mais de uma palavra (com espaços em branco)

**gets:** esta função é utilizada para ler strings do STDIN (normalmente o teclado). **gets** continua lendo caracteres até que NEWLINE ou EOF seja encontrado.

Sintaxe:

```
char nome[80];
gets (nome);
```

Observação: **gets não verifica o tamanho do buffer** de entrada de dados. Por este motivo, pode acabar sobrescrevendo informações anteriormente armazenadas nesta área de memória e corromper o funcionamento do software e até mesmo do sistema.

**fgets:** é usado para ler linhas de dados. Pode ser utilizado tanto para leituras de arquivos como para entradas pelo teclado. Deve ser utilizado preferencialmente no lugar de **gets**, pois **fgets** verifica se o dado que está sendo lido não excede o tamanho do buffer.

Sintaxe (para leituras de strings pelo teclado):

```
char sNome[80];
fgets (sNome, sizeof(sNome), stdin);
```

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

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

➔ **Por que se está acessando a segunda letra e não a primeira?** Na linguagem C, o índice de vetores 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 "pera". Em seguida, ele mudará esta letra e apresentará a string no final.

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

Nesta string (" 'p' 'e' 'r' 'a' '\o' "), 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 'p' 'e' 'r' 'a' e '\0' simplesmente declarando `char str[10] = "pera"`.

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

## 12.2 Exercícios com strings (típicos de prova)

**1)** Escreva um programa que leia duas strings do teclado e as imprima na tela. Imprima também a segunda letra de cada string.

**2)** Faça um programa que permita que o usuário digite uma string (com espaços) de até 80 caracteres. O programa deve contar quantos caracteres **'a'** existem dentro da string digitada.

**3)** O mesmo programa anterior, mas agora o programa deve permitir que o usuário digite também o caractere a ser encontrado. E o programa deverá contar quantos caracteres (do solicitado pelo usuário) existem dentro da string também digitada pelo usuário.

*Exemplo:*

- Digite uma string com até 80 caracteres: blablabla

- Digite um caractere que deseja encontrar na string acima digitada: a
(saída do programa) = Existem 3 caracteres 'a' na string "blablabla".

# 13 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: 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 4.1**. 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 exemplo é que **main()** é uma função especial, cujo diferencial é o fato de ser a primeira função a ser executada em um programa.

## 13.1 Argumentos (Parâmetros)

Argumentos, ou parâmetros, são as entradas que a função recebe. É através dos argumentos que passamos informações 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 quadrado (int iArgumento) /* Calcula o quadrado de x */
 {
    printf ("O quadrado eh %d",(iArgumento * iArgumento));
    return(0);
 }
 int main ()
 {
    int iNum;
    printf ("Entre com um numero: ");
    scanf ("%d",&iNum);
    printf ("\n\n");
    quadrado(iNum);
    return(0);
 }
```

Na definição de `quadrado()` dizemos que a função receberá um argumento inteiro `iArgumento`. Quando fazemos a chamada à função, o inteiro `iNum` é passado como argumento.

**Há alguns pontos a observar:**

- Em primeiro lugar, temos que satisfazer aos requisitos da função quanto ao tipo e à quantidade de argumentos quando a chamamos. Apesar de existirem algumas conversões de tipo, que a linguagem 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 `iNum`, ao ser passada como argumento para `quadrado()` é copiada para a variável `iArgumento`. Dentro de `quadrado()` trabalha-se apenas com `iNum`. Se mudarmos o valor de `iNum` dentro de `quadrado()` o valor de `iNum` na função `main()` permanece inalterado.

Vamos ver um exemplo de **função que receba mais de uma variável como argumento**. 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=2;
   y=3;
   mult (x,y,4);
   return(0);
}
```

## 13.2 Retornando Valores

Muitas vezes é necessário fazer com que uma **função retorne um valor ao local onde ela foi chamada**. 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 a seguir:

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

int main ()
{
  int iSaida;
  iSaida=prod (12,3);
  printf ("A saida eh: %d\n",iSaida);
  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 `iSaida`, que

posteriormente foi impressa usando o `printf`. Uma observação adicional: se não especificarmos o tipo de retorno de uma função, o compilador C automaticamente supõe que esse tipo é **inteiro**. Então, não é uma boa prática não especificar o valor de retorno.

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 fSaida;
   fSaida=prod (2.5,4);
   printf ("A saida eh: %f\n",fSaida);
   return(0);
}
```

## 14 Auto Avaliação: 2ª. Parte da Disciplina

➔ Veja como você está: escreva um programa que leia dois números (um inteiro e um ponto flutuante) na função principal. Estes valores deverão ser passados para uma função que some estes dois valores lidos do teclado. Esta função deverá retornar para uma variável da função principal o resultado da soma. Imprimir o conteúdo desta variável na função principal.

## Referências Bibliográficas

Para elaboração desta apostila foram consultados vários tipos de materiais, como: livros, outras apostilas, páginas web etc. Algumas das referências consultadas estão apresentadas neste capítulo, mas grande parte do material disponibilizado na apostila, como exemplos e exercícios foram fornecidos pelo **Prof. Marcelo Azambuja**. Esse material é utilizado por ele em suas aulas de Programação II – C e está disponível em: *http://professores.faccat.br/azambuja/*

[CRU97] CRUZ, Adriano Joaquim de Oliveira. **Material de aula sobre Linguagem C.** Disponível em: http://equipe.nce.ufrj.br/adriano/c/home.htm. Acesso em: Mar. 2008.

[MAN97] MANZANO, José Augusto N. G. **Estudo Dirigido de Linguagem C.** São Paulo, Érica, 1997.

[OUA97] OUALLINE, Steve. **Practical C Programming - 3rd. Edition**. Editora O'REILLY, 1997. Disponível em: http://www.oreilly.com/catalog/pcp3/chapter/ch13.html. Acesso em: Mar. 2008.

[SCH97] SCHILDT, Herbert. **C - Completo e Total – 3ª. Edição.** São Paulo, Pearson Makron Books, 1997.

[ZIV02] ZIVIANI, Nivio. **Projeto de Algoritmos: com implementações em Pascal e C**. São Paulo, Pioneira Thomson Learning, 2002.