Apostila de Linguagem C

Prof. Francesco Artur Perrotti

Fatec Americana

Introdução

No início da década de 70 nos laboratórios da *Bell Telephones Inc*, Ken Thompson criou a linguagem B a partir da linguagem BCPL, então em uso pela empresa. Mais tarde, em 1978, Dennis Richie recebeu a ajuda de Brian Kerninghan para aprimorar a linguagem B. O resultado foi chamado de linguagem C. Pelas suas características de portabilidade e estruturação a nova linguagem muito rapidamente se torna popular entre os programadores, tanto que apenas dois anos depois de sua criação ela foi padronizada pelo *American National Standard Institute*, surgindo assim o **ANSI C**.

Em 1992, os teóricos do desenvolvimento da teoria de Programação Orientada a Objetos (*Object Oriented Programming – OOP*) concordaram em utilizar a linguagem C para a implementação da OOP, surgindo então a linguagem C++ (C orientado a objeto).

Tipos de linguagens de programação

As linguagens de programação, podem ser classificadas em linguagens de **baixo ou alto nível**. Entende-se por linguagem de **baixo nível** as que são desenvolvidas para serem entendidas pelo processador do computador, são genericamente chamadas de linguagens *Assembly*. Programas feitos nessas linguagens são extremamente rápidos, pequenos e eficientes, já que as instruções são colocadas diretamente em linguagem de máquina. A desvantagem dos programas feitos em Assembly é que eles têm pouca *portabilidade*. Um código gerado para um processador não serve para outro. Além disso, códigos Assembly não são muito legíveis para humanos, tornando a programação mais difícil.

As linguagens de alto nível são linguagens voltadas para o ser humano. Em geral utilizam estruturas em sua sintaxe, tornando o código muito mais legível. Porem, para que o programa possa ser executado pelo computador, ele precisa ser convertido para o Assembly do processador. Esse processo de conversão pode ser feito de duas maneiras. A primeira é através de um **Interpretador**, que é um programa que lê as instruções escritas pelo programador **(código fonte)**, converte em linguagem de máquina e envia o código já convertido para execução pelo processador. A outra maneira é através de um **Compilador**, esse programa lê o arquivo onde está o código fonte, converte as instruções para linguagem de máquina e grava o código já convertido em outro arquivo que pode ser diretamente executado sem nenhuma conversão (**código executável**).

Linguagem C

A **linguagem C** é uma linguagem genérica de alto nível. Foi desenvolvida *por* programadores *para* programadores tendo como meta características de flexibilidade e portabilidade. Entre suas principais características, podemos citar:

- **C** é uma linguagem de alto nível com uma sintaxe bastante estruturada e flexível tornando sua programação bastante simplificada.
- Programas em **C** são compilados, gerando programas executáveis.
- **C** compartilha recursos tanto de alto quanto de baixo nível, pois permite acesso e programação direta do microprocessador. Com isto, rotinas cuja dependência do tempo é crítica podem ser facilmente implementadas usando instruções em Assembly. Por esta razão o **C** é a linguagem preferida dos programadores de aplicativos.

- O **C** é uma linguagem estruturalmente simples e de grande portabilidade. O compilador C gera códigos mais enxutos e velozes do que da maioria das outras linguagens.
- Embora estruturalmente simples (poucas funções intrínsecas) o **C** não perde funcionalidade, pois permite a inclusão de uma farta quantidade de rotinas do usuário. Os fabricantes de compiladores fornecem uma ampla variedade de rotinas pré-compiladas em bibliotecas.

Iniciando em C

Como citado, C é uma linguagem que foi feita para programadores, o objetivo era criar uma linguagem eficiente e poderosa que pudesse substituir o assembly na criação de uma nova versão do sistema Unix. Sem dúvida que esse objetivo foi atingido, porém aprender C é preciso entender alguns conceitos de programação que não são tão básicos.

C é sensível ao caso das letras

A primeira coisa que devemos manter em mente ao escrever um programa em C é que, a linguagem C é sensível ao caso das letras. Por exemplo, X e x são nomes diferentes para o compilador do C e podem indicar duas variáveis diferentes. É claro que um bom programador vai evitar o uso de nomes tão parecidos para duas variáveis diferentes, mas isso é possível. Da mesma forma, os comandos e funções padrão em C também devem ser invocados com o caso correto, do contrário não serão reconhecidos pelo compilador. Todos os comandos e funções padrão do C são escritos inteiramente com letras minúsculas.

A função main

Todo programa escrito em C deve obrigatoriamente ter uma e apenas uma função chamada **main**. Esta função será o ponto inicial do programa, é a partir dela que todas as outras funções serão chamadas.

Assim como todas as funções em C, a função **main** também pode retornar um valor. No caso específico da função **main**, se ela retornar um valor, este deve ser um número inteiro e será retornado para o sistema operacional. A palavra reservada **return** é utilizada (em qualquer função) para determinar o valor de retorno. Caso a função não tenha nenhum valor de retorno ela deve ser declarada com o tipo **void**, que indica que não há valor de retorno. Se a palavra **void** for utilizada na declaração de parâmetros da função, indica que a função não recebe parâmetros.

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

No exemplo acima estamos indicando que a função **main** não receberá nenhum parâmetro e não retornará nenhum valor. É importante lembrar que o tipo default para o retorno de qualquer função é o inteiro:

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

Agora como o tipo da função foi omitido, será associado o tipo inteiro (**int**) ao retorno da função. O uso de **void** nos parâmetros para indicar que não há parâmetros é opcional.

Delimitadores

Os delimitadores essenciais para iniciar a programação em C são os seguintes:

{ } Chaves: delimitam um bloco de comandos.

( ) Parêntesis: delimitam a lista de parâmetros das funções.

“ Aspas: delimitam cadeias de caracteres (strings).

‘ Apóstrofe: delimitam um único caracter.

[ ] Colchetes: delimitam índices em vetores ou matrizes.

/* */ Abre e fecha comentários no programa.

Inclusão de bibliotecas

C é uma linguagem bastante enxuta, a definição do ANSI C utiliza apenas 32 palavras reservadas para a linguagem. Não fazem parte dessa lista, por exemplo, todos os comandos de entrada e saída. Toda a parte de entrada e saída na linguagem C está em bibliotecas de funções que podem ser incluídas nos programas. Boa parte dessas bibliotecas de funções está definida no padrão ANSI e portanto podem ser encontradas em todos os compiladores C. Também é possível e bastante simples criar novas bibliotecas e incluí-las nos programas. Essa é uma prática comum na programação em C, além de ser também aconselhável, porque permite o aproveitamento do código já escrito.

Para incluir uma biblioteca de funções em um programa basta utilizar a diretiva `#include` seguida do nome do arquivo cabeçalho (header) da biblioteca entre os sinais de menor e maior, como no exemplo abaixo:

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

Normalmente essa diretiva é utilizada no início do programa, antes das funções, mas pode ser colocada em qualquer ponto do programa. A diretiva `#include`, assim como todas as diretivas iniciadas com o símbolo #, é tratada pelo pré-processador do compilador. Em resumo o que acontece quando essa diretiva é encontrada, é a inclusão do arquivo indicado no local da diretiva. Portanto em tese, a diretiva pode ser usada em qualquer ponto do programa, porém se o arquivo indicado é um arquivo de cabeçalho, caso mais comum, então a diretiva não deve estar dentro de nenhuma função.

Identificadores

Identificadores são os nomes usados para se fazer referência a variáveis, funções, rótulos, constantes e outros objetos definidos pelo usuário. Assim como na maioria das linguagens o primeiro caracter do identificador tem que ser uma letra (a...z ou A...Z) ou um sublinhado (_). Os caracteres seguintes podem ser letras, números, sublinhado ou até um cifrão ($).

Importante lembrar novamente que C é sensível ao caso das letras, por exemplo **`nalunos`**, **`Nalunos`**, **`NAlunos`** e **`NALUNOS`** são nomes diferentes para o C e podem identificar objetos diferentes. É claro que um bom programador não utilizaria essa característica para nomear variáveis diferentes, simplesmente porque o programa ficará confuso pra nós humanos.

Tipos de dados

C tem apenas 5 tipos básicos de dados, listados a seguir:

- **`int`**: valor inteiro com sinal (pode assumir valores negativos)
- **`char`**: caracter ou inteiro com 1 byte
- **`float`**: valor real
- **`double`**: valor real com dupla precisão
- **`void`**: sem tipo/sem valor

As variáveis do tipo **`char`**, podem ser usadas tanto para representar caracteres (como 'a', 'A', 'b', etc) ou valores inteiros de 8 bits (de 0 a 255 ou de –128 a +127). O tipo **`void`** indica que não há valor associado. É comum ser usado em funções para indicar que a função não retorna nenhum valor.

Com exceção de **`void`**, os outros tipos aceitam ser precedidos por modificadores que alteram suas características, permitindo uma maior flexibilidade do programa e variedade de tipos.

Modificadores:

	Tipos modificados	Significado
`Unsigned`	**`int`** e **`char`**	Sem sinal – não assume valores negativos
`Signed`	**`int`** e **`char`**	Com sinal – assume valores negativos
`Long`	**`int`** e **`double`**	Pode dobrar (ou não) o número de bytes usados para armazenar o valor
`Short`	**`int`**	Curto

A tabela abaixo mostra as combinações possíveis de tipos e modificadores no Turbo C e a faixa de valores que os tipos modificados podem assumir:

Tipo modificado	No. de bytes	Valor mínimo	Valor máximo
`Char`	1	0	255
`signed char`	1	-128	127
`unsigned char`	1	0	255
`int`	2	-32768	32767
`signed int`	2	-32768	32767
`unsigned int`	2	0	65535
`short int`	2	-32768	32767
`unsigned short int`	2	0	65535
`long int`	4	-2147483648	2147483647

signed long int	4	-2147483648	2147483647
unsigned long int	4	0	4294967295
float	4	$3.4x10^{-38}$	$3.4x10^{+38}$
double	8	$1.7x10^{-308}$	$1.7x10^{+308}$
long double	8	$1.7x10^{-308}$	$1.7x10^{+308}$

Observação importante: **int** é sempre o tipo default quando o tipo de dado é omitido, isso vale para funções, variáveis e demais objetos tipados. No exemplo abaixo, as variáveis **i1** e **i2** são exatamente do mesmo tipo, assim como as variáveis **lg1** e **lg2**:

```
unsigned i1;
unsigned int i2;
long lg1;
long int lg2;
```

Declaração de variáveis

A forma geral de um comando de declaração de variáveis em C é a seguinte:

[modificador(es)] tipo lista_de_variáveis;

Se houver várias variáveis na lista, elas devem ser separadas por vírgulas, como no exemplo abaixo:

```
int n1, n2, p1, p2;
long l1, l2;
unsigned long u1, u2, u3;
double f1,f2;
```

Existem três lugares onde as variáveis podem ser declaradas. O primeiro é fora de todas as funções, inclusive da função main( ). Essas variáveis são chamadas de **variáveis globais** e ficam visíveis para o programa inteiro a partir do ponto onde foi declarada. As variáveis globais têm sua utilidade na programação e podem facilitar bastante a vida do programador se forem bem usadas, porém como regra geral deve-se evitar ao máximo seu uso, porque elas podem ser responsáveis pela propagação de erros nos programas. Sempre que possível utilize **variáveis locais**. O segundo lugar onde elas podem ser declaradas é dentro de uma função. Nesse caso, elas são declaradas no início da função, antes dos comandos. Essas são as **variáveis locais**. Ficam visíveis apenas para os comandos da função onde ela foi declarada. As outras funções não têm acesso a essas variáveis. O último lugar é na declaração de parâmetros das funções, nesse caso elas são utilizadas para receber os valores dos parâmetros da função quando esta é chamada de outra parte do programa. A partir daí os parâmetros funcionam como qualquer variável local.

Inicialização de variáveis

Em C a maioria das variáveis locais e globais pode ser inicializadas no momento da sua declaração. Esse é um recurso que torna o programa menor e mais eficiente. Para inicializar uma variável basta colocar um sinal de igual após o nome da variável e a seguir o valor que ela deve assumir inicialmente. A forma geral é:

[modificador(es)] tipo nome_da_variável = constante;

Exemplos:

```
char c = 'a';
long l1 = 70500;
int  n=0, d=10;
char st[20] = "exemplo";
```

Importante: As variáveis globais são inicializadas apenas no início da execução do programa, já as locais são inicializadas cada vez que a função onde ela está declarada é executada.

Constantes

Constantes são valores fixos que o programa não pode alterar. Elas podem ser de qualquer tipo básico de dados. A maneira de representar uma constante depende do seu tipo. Para uma constante do tipo char, devem ser usados apóstrofes delimitando o caracter. Constantes de strings utilizam aspas. Constantes inteiras são representadas sem o ponto decimal, já as constantes reais (float ou double) exigem o ponto decimal. Veja os exemplos:

```
'a'             char
10              int
-100            int
10.0            float
"teste"         char (string)
2.1e50          double
-0.7293876134   double
```

Constantes octais e hexadecimais

Algumas vezes é mais fácil trabalhar com os números se eles estiverem em um sistema numérico de base 8 (octal) ou de base 16 (hexadecimal). As constantes octais sempre começam por 0 (zero) e só podem utilizar os dígitos de 0 a 7. As constantes hexadecimais sempre começam por 0x (zero xis) e utilizam os dígitos de 0 a 9 e as letras de A a F, que representam os números de 10 a 15, respectivamente. Veja os exemplos:

`011`	octal (9 em decimal)
`020`	octal (16 em decimal)
`0xFF`	hexadecimal (255 em decimal)
`0x10`	hexadecimal (16 em decimal)

Mesmo que você não use constantes octais, é importante saber como são declaradas por que se por um motivo qualquer for utilizado um zero antes de um número o compilador pode interpretar esse número como octal e o programa pode produzir resultados diferentes do esperado.

Constantes com barra invertida

O uso de apóstrofes para envolver as constantes de caracteres funciona para a maioria de caracteres imprimíveis, porem, existem alguns caracteres que não podem ser inseridos pelo teclado, como ENTER por exemplo, que geralmente tem a função de encerrar a edição de um campo ou de mudar de linha em um texto. Para representar esses caracteres dentro do programa o C fornece as constantes com barra invertida. Segue abaixo a lista dessas constantes:

Código	Significado
`\b`	Retrocesso (back space)

\f	Alimentação de formulário
\n	Nova linha
\r	Retorno de carro (inicio da linha)
\t	Tabulação horizontal
\v	Tabulação vertical
\"	Aspas
\'	Apóstrofe
\0	Valor zero
****	Barra invertida
\a	Alerta (beep)

Operadores

Apesar de ser uma linguagem bastante enxuta em palavras reservadas, C é rico em operadores. Um operador é um símbolo que informa ao compilador que ele deve realizar uma operação lógica ou matemática. C tem 3 categorias de operadores: os *aritméticos*, *relacionais/lógicos* e *bit a bit*. Além disso, existem operadores especiais para tarefas particulares.

Operadores aritméticos

A tabela abaixo lista os operadores aritméticos em C:

Operador	Operação
+	Adição
-	Subtração ou menos unário (inversão de sinal)
*****	Multiplicação
/	Divisão
%	Resto da divisão inteira
--	Decremento (unário)
++	Incremento (unário)

Para os operadores de soma, subtração e multiplicação não há novidade, eles funcionam exatamente como na maioria das outras linguagens. Porém, o operador de divisão funciona de maneira diferente. Existem dois operadores de divisão: inteira (a que pode deixar resto) e divisão real, onde o resultado pode ter casas decimais. Para as duas operações é utilizado o mesmo operador (/). C irá realizar uma divisão inteira ou real conforme o tipo dos operandos envolvidos. Se os dois termos da operação são inteiros (char, int ou long int) a divisão será inteira. Se pelo menos um dos termos for de ponto flutuante (float ou double), então a divisão será real. Veja os exemplos abaixo:

10 / 3 → Divisão inteira (os dois termos são inteiros) o resultado será 3
10 / 3.0→ Divisão real (3.0 é considerado ponto flutuante) o resultado será 3.33333...

O operador de resto (%) só pode ser utilizado com operandos inteiros, nenhum dos termos pode ser float ou double.

Casts

Colocar um ponto decimal em um valor numérico faz com que ele seja considerado como um valor real. Esse artifício é bastante útil para garantir que as operações sejam realizadas como desejado, mas nem sempre é possível utilizá-lo. Veja o exemplo abaixo onde x é float e y e z são variáveis inteiras:

```
float x;
int y, z;
. . .
x = y / z;
```

Apesar de x ser float, y e z sendo variáveis inteiras, a divisão será também inteira e o resultado será convertido para o tipo float apenas no momento da atribuição a x, já que este é float, mas quando isso ocorrer, a divisão já terá sido realizada como uma divisão inteira. Se for desejado que a divisão seja realizada como uma divisão real, é necessário que um dos dois termos da divisão seja convertido para um valor real, antes que a divisão seja efetuada. O modo de fazer isso é utilizando o recurso de **cast** que o C oferece. Com esse recurso é possível converter um valor ou variável de um tipo em outro tipo. Para isso basta colocar o tipo desejado entre parêntesis antes do valor ou variável ou expressão que deve ser convertido. Exemplo:

```
x = (float) y/z;
```

Neste exemplo a variável y está sendo convertida para o tipo float antes que a operação seja realizada, portanto a divisão será real. Note que essa conversão vale apenas para a expressão onde o cast aparece, a variável y continua sendo uma variável inteira, se for utilizada em outros comandos ou expressão será como uma variável inteira.

Importante: O cast será aplicado à variável ou expressão que estiver logo depois. Se nesse exemplo fossem usados parêntesis na divisão o resultado não seria o esperado. Veja:

```
x = (float)(y/z);
```

Agora o que será convertido para float, não é a variável y, mas o resultado da divisão, que será uma divisão inteira.

Incremento e decremento

Estes dois operadores que C oferece são muito utilizados e tornam o programa executável muito mais eficiente. O operador ++ acrescenta 1 ao valor do operando e o operador `--` subtrai 1, estas operações são muito comuns na programação.

Supondo uma variável `x` do tipo `int`, as seguintes operações são equivalentes:

`x++;` é equivalente a `x = x + 1;` e
`x--;` é equivalente a `x = x - 1;`

Apesar de equivalentes, o uso dos operadores de incremento e decremento gera um código executável menor e mais eficiente e portanto deve ser preferido. Os dois operadores poder ser colocados antes ou depois do operando, de qualquer maneira o operando será incrementado ou decrementado conforme o caso. A diferença existe quando esse operando faz parte de uma expressão maior. Nesse caso se o operador está colocado antes do operando, a operação de incremento/decremento é efetuada antes de a expressão ser resolvida, se estiver colocado depois do operando, então, primeiro a expressão é resolvida, depois é feita a operação de incremento/decremento. Veja o exemplo abaixo onde x e y são do tipo `int`:

```
x = 10;
y = ++x;
```

Neste caso, como o operador de incremento vem antes do operando, então primeiro x é incrementado, depois a expressão é resolvida, sendo assim y terminará com o valor 11. Vejamos o outro caso:

```
x = 10;
y = x++;
```

Agora o operador está depois do operando, então primeiro a expressão é resolvida, depois x é incrementado, portanto y termina com o valor 10. Veja que nos dois casos x termina com 11, a diferença é quando x é ajustado para 11, antes ou depois de atribuir o valor para y.

Operadores Relacionais e Lógicos

Como todas as linguagens genéricas, C também oferece um conjunto completo de operadores relacionais e lógicos. Operadores relacionais são utilizados para estabelecer relações entre dois valores, ou em outras palavras, para comparar dois valores. Os operadores lógicos permitem realizar operações lógicas (**!**). Veja abaixo a tabela de operadores relacionais e lógicos oferecidos pela linguagem C:

Operadores relacionais

Operador	Relação
>	Maior
<	Menor
>=	Maior ou igual
<=	Menor ou igual
= =	Igual
!=	Diferente

Operadores Lógicos

Operador	Operação
&&	E (and)
||	Ou (or)
!	Negação – unário (not)

Operador de atribuição

Em C, o operador de atribuição é o sinal de igual (=). Assim como em outras linguagens, este operador faz com que a expressão do lado direito dele seja resolvida e o resultado seja atribuído à variável que estiver do lado esquerdo.

```
variável = expressão;
```

No entanto em C, a operação de atribuição também provoca um retorno do valor que foi atribuído e pode ser utilizado em outras expressões. Por exemplo, considere o trecho abaixo:

```
int x, y, z;
```

```
x = y = z+10;
```

Veja que há duas atribuições nesta expressão. C irá calcular o valor de `z+10` e atribuirá o resultado à `y`, no entanto, essa atribuição gera um retorno (o valor que foi atribuído a `y`) e esse retorno será atribuído à variável `x`. Portanto no final `x` também terá o mesmo valor que `y`. É muito comum o uso dessa característica de C para zerar várias variáveis na mesma linha de comando, como no exemplo abaixo:

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

Apesar de ser um código um tanto confuso, principalmente pra quem está iniciando em C, é possível fazer o retorno da atribuição participar de expressões mais complexas do que simplesmente atribuir o mesmo valor a várias variáveis. Veja o exemplo abaixo:

```
x= (y= (z= 20) * 2) + 4;
```

Neste exemplo, primeiro `z` receberá o valor 20, o retorno dessa atribuição (20) será multiplicado por 2 e o resultado atribuído a `y`. Em seguida o retorno da atribuição a `y` (40) será somado a 4 e atribuído a `x`. Note o uso dos parêntesis para garantir que a operação será realizada como desejado. A linha acima é equivalente ao código abaixo:

```
z = 20;
y = z * 2;
x = y + 4;
```

Combinando a atribuição com operadores aritméticos

C também permite que o operador de atribuição seja combinado com outros operadores para que a variável que está recebendo a atribuição participe da expressão à direita. Por exemplo, ao combinar o operador de soma com o de atribuição, o efeito é um operador de auto-incremento, onde o valor da variável destino será utilizado na operação de soma:

```
x += 10;
```

é equivalente a:

```
x = x + 10;
```

Da mesma maneira os outros operadores aritméticos também podem ser combinados com a atribuição:

```
x -= 10;   equivale a:   x = x - 10;
x *= 10;   equivale a:   x = x * 10;
x /= 10;   equivale a:   x = x / 10;
```

Atribuição de strings

Strings não podem ser atribuídas diretamente como as variáveis numéricas ou caracteres. Para atribuir um valor string em uma variável é necessário utilizar a função `strcpy()` que está declarada em `strings.h`. Veja o exemplo abaixo:

```
# include <string.h>

void main () {
   char St[20];
   strcpy (St, "Valor string");
}
```

Entrada e saída do console

Console é o conjunto formado pelo teclado e monitor de vídeo. As entradas e saídas de dados utilizam como default o console. As bibliotecas C oferecem um conjunto de funções bastante completo para entrada e saída, divididas em 3 categorias: funções de E/S de caracteres, de strings e de uso geral.

Entrada de caracteres

char getche ( )

Esta função espera que seja pressionada uma tecla, ecoa (escreve) o caracter na tela e retorna sem esperar que a tecla ENTER seja pressionada.

Exemplo:

```
char ch;
ch = getche();
```

Variações:

char getch ( ) funciona da mesma maneira que `getche()`, mas não ecoa o caracter lido na tela.

char getchar ( ) usa um buffer de entrada para ler os caracteres. Exige o ENTER para retornar. Não é muito utilizada atualmente, existe mais para fins de compatibilidade com programas mais antigos.

Saída de caracteres

putchar (*caracter*); Escreve na tela o caracter passado como argumento (só trabalha c/ caracteres, não use para strings).

Exemplo:

```
main(){
  char  c;
  c = getch (); // ou
  putchar (c);  // putchar (getch());
}
```

Entrada e saída de strings

C não define um tipo específico para as strings, o que existe são vetores de caracteres terminados com um zero. Esse formato permite grande flexibilidade porque não impõe um tamanho máximo para as strings. A string pode ter qualquer tamanho, desde que o último caracter seja um zero. Todas as funções que trabalham com strings esperam que elas estejam nesse formato. A biblioteca **`strings.h`** fornece um conjunto bastante completo de funções de suporte a strings.

gets (*variável*); lê uma string do teclado.

A função **`gets()`** retorna um ponteiro para a variável que foi utilizada para leitura, mas esse retorno pode ser ignorado sem problemas. No caso de gets ( ) a leitura é feita até que seja pressionada a tecla ENTER. O ENTER não é incluído na string lida, mas é acrescentado automaticamente um zero no final da string como terminador.

A função **`gets()`** também coloca automaticamente um '\n' na tela ao final da leitura, ou seja, após a leitura o cursor vai para a linha de baixo.

puts (*string*); escreve uma string na tela.

A função **puts()** também aceita os caracteres de controle que são escritos com barra invertida.

Exemplo:

```
main(){
  char nome [51];
  puts (”digite seu nome: ”);
  gets (nome);
  puts(”\nSaudações ”);
  puts(nome);
}
```

Comandos genéricos de entrada e saída

Função printf()

Permite fazer a saída formatada de qualquer tipo básico de dados.

Sintaxe:

printf (*string_de_controle, lista_de_argumentos*>);

A função printf() pode ser usada para mostrar na tela qualquer tipo de dados de acordo com o que foi especificado na string de controle. As referências aos dados na lista de argumentos são feitas usando uma seqüência de caracteres que sempre iniciam com o caracter porcento (%). Para cada referência na string de controle é necessário existir um argumento na lista de argumentos. Os argumentos devem ser colocados na lista na mesma ordem em que são referidos na string de controle.

String de controle

Pode conter comandos de formatação e também outros caracteres. Tudo que não for comando de formatação será impresso na tela exatamente como aparece na string de controle.

Comandos de formatação

Seguem sempre a seguinte sintaxe:

% [-] [especificação de tamanho] [modificador] *tipo*

O alinhamento default para a função printf é sempre à direita. Se for necessário que seja feito um alinhamento à esquerda, então o sinal de menos (-) deve ser colocado após o símbolo %.

Especificação de tamanho

Um valor inteiro colocado após o % ou (quando existir) o sinal (-), indica o tamanho mínimo que o dado deve ocupar.

Exemplo: %10d

Este comando imprime um valor inteiro, que ocupará o espaço de 10 caracteres na tela e será alinhado à direita. O mesmo vale para strings, caracteres ou valores com ponto flutuante.

Exemplo: %10s
%10f

No caso de strings ou inteiros também pode ser definido o tamanho máximo que será utilizado para imprimir o dado na tela. Para definir o tamanho máximo, coloque um ponto após o tamanho mínimo e em seguida o tamanho máximo.

Exemplo: `%10.15s`

Neste caso, será impressa uma string que ocupará no mínimo 10 caracteres e no máximo 15 caracteres. Se a string tiver mais que 15 caracteres, ela será truncada. O mesmo vale para valores inteiros.

Para números com ponto flutuante (float ou double) o valor depois do ponto indica o número de casas decimais que deve ser impressas.

Modificadores

São utilizados para indicar que um valor numérico é long ou short. A letra `l` indica long e a letra `h` indica short. O long aplicado ao tipo `f` (float) indica que o valor é double.

Ex: **`%li`** (long int)
`%hi` (short int)
`%lf` (double)

Tipos aceitos:

`i` ou `d`	inteiro
`x`	inteiro em hexadecimal
`o`	inteiro em octal
`s`	string
`c`	caracter
`u`	unsigned (inteiro sem sinal)
`f`	ponto flutuante
`e`	ponto flutuante em notação científica (1.777e3 = 1.777×10^3)
`g`	usa e ou f, o que ocupar menos espaço
%	imprime o caracter %

Exemplo:

`%-8.2lf` Imprime valor tipo double que ocupará 8 espaços, será impresso com 2 casas decimais e será alinhado à esquerda.

Função `scanf()`

Leitura de qualquer tipo básico de dados definido pela linguagem C.

Sintaxe:

`scanf(`*`string de controle,lista de endereços de variáveis`*`);`

É possível a leitura de mais de uma variável na mesma chamada. Para cada variável lida deve haver na string de controle um código de formatação correspondente. Os códigos devem aparecer na string de controle exatamente na mesma ordem que as variáveis que serão lidas.

Endereços das variáveis

Diferente da função `printf()` que pode trabalhar com variáveis ou valores constantes, a função `scanf()` precisa modificar o conteúdo das variáveis fornecidas para poder armazenar os valores lidos nelas. Para isso é preciso que as variáveis sejam passadas à função por referência e não por valor. Significa que o que a função precisa realmente é uma referência à variável, ou seja, o endereço dela. Essa referência pode ser fornecida através de ponteiros, ou então é possível utilizar o operador **&** fornecido pelo C que retorna o endereço da variável que estiver logo após. Assim, sempre que for feita uma leitura de variáveis inteiras, de ponto flutuante ou caracteres é necessário colocar o operador **&** antes da variável dentro da chamada à função `scanf()`. Com variáveis strings isso não é necessário porque uma string é na verdade um vetor de caracteres e todo vetor em C já é naturalmente um ponteiro.

String de controle:

A string de controle pode conter:

Códigos de formatação	→	iniciam com %
Caracteres brancos	→	espaço, tabulação e `\n`
Caracteres não brancos	→	qualquer outro caracter

Códigos de formatação

Funcionam de modo parecido com os códigos de formatação da função `printf()`. Também iniciam obrigatoriamente com o caracter % (porcento) e tem que indicar um tipo de dados no final. Entre o % e o tipo podem aparecer campos opcionais.

Sintaxe:

`%[*][Tam Máx][Modificador]`*`tipo`*

Tipos aceitos:

`i`	inteiro (em decimal, hexadecimal ou octal)
`d`	inteiro em decimal
`x`	inteiro em hexadecimal
`o`	inteiro em octal
`s`	string
`c`	caracter
`u`	unsigned (inteiro sem sinal)
`f` ou `g`	ponto flutuante
`e`	ponto flutuante em notação científica (1.777e3 = 1.777x10³)
%	lê o caracter %

Modificadores:

`h`	curto
`l`	longo

O modificador `l` aplicado a inteiros significa que será lido um valor do tipo long int. Aplicado a float (`f`) significa que será lido um valor do tipo double. Note que não há o tipo double entre os tipos aceitos. Para ler um valor double é preciso utilizar o modificador `l` junto com o tipo `f` (`%lf`).

Tamanho máximo

Limita a leitura de dados ao número de caracteres indicado. No caso de strings ou inteiros, este campo fará com que apenas o número indicado de caracteres seja armazenado na leitura do campo.

Se durante a digitação forem digitados mais caracteres do que o exigido pelo campo, os caracteres adicionais ficarão armazenados na stream de entrada (`stdin`) e serão utilizados na(s) próxima(s) leitura(s) de dados. Note que este campo não limita a quantidade de caracteres que podem ser digitados, limita apenas a quantidade de caracteres que serão considerados para a leitura. Se for digitado um caracter branco (veja abaixo), a leitura do campo será encerrada mesmo que o limite ainda não tenha sido atingido.

Exemplo:

```
int Num;
char st[20];
scanf("%2i", &Num);
scanf("%s", st);
```

Neste exemplo, a leitura do inteiro está limitada a 2 caracteres. Se a entrada digitada for algo como `12345` para o inteiro, apenas os dois primeiros dígitos (`12`) serão considerados para a variável `Num`. Os outros 3 dígitos serão utilizados para a leitura da variável `st` e ela conterá o valor `"345".`

*** Para descartar campos**

Um ***** colocado logo após o **%**, indica que aquele campo deve ser lido, mas não deve ser armazenado. Portanto não deve haver nenhuma variável na lista de variáveis para a leitura de campos com ***** no código de formatação.

Caracteres brancos e não brancos

Caracteres brancos colocados na string de controle indicam que serão usados caracteres brancos para separar os campos na leitura. Estes caracteres (espaço, tabulação e `\n`) são intercambiáveis, não há diferença entre utilizar um ou outro.

Os caracteres não brancos também funcionam como separadores de campos, mas exigem que durante a entrada seja digitado exatamente o que aparece na string de controle. Por exemplo, se x e y forem variáveis inteiras, então a string de controle:

```
scanf("%i,%i",&x,&y);
```

Exige uma entrada do tipo `10,20`, com a vírgula digitada logo após o primeiro número. Note que se a string de controle for:

```
scanf("%i%i",&x,&y);  ou scanf("%i  %i",&x,&y);
```

Então o usuário teria que usar um caracter branco para separar os dois inteiros.

A entrada padrão do C – `stdin`

A função `scanf()` utiliza a **stream** de entrada padrão do C para a leitura de dados. O conceito de stream no C é o de uma fila de bytes que independe de dispositivo, de forma que diferentes dispositivos podem ser tratados da mesma maneira quando há uma stream associada a eles. O que é importante saber para utilizar bem a função `scanf()` é que eventualmente após uma leitura de dados podem sobrar caracteres não utilizados. Esses caracteres ficam armazenados na stream `stdin` e serão utilizados na próxima leitura, o que pode provocar efeitos inesperados no programa. Para evitar esses efeitos existe a função `fflush()`, incluída na biblioteca `stdio.h` que esvazia uma stream. É aconselhável o uso dessa função antes de utilizar a função `scanf()` para garantir que não será utilizado nenhum caracter que sobrou de uma leitura anterior na leitura do campo atual.

Sintaxe:

```
fflush( <stream> );
```

Exemplo:

```
fflush( stdin );
```

Comandos de controle de fluxo

Estes comandos controlam o fluxo de execução do programa através de desvios e repetições. Todos eles fazem parte da linguagem e utilizam palavras reservadas que não podem ser utilizadas como identificadores de variáveis ou funções.

Há 3 categorias em que podem ser divididos os comandos de controle de fluxo:

1. Desvio condicional
   - **`if(), if()-else`**
   - **`switch()`**
2. Desvio incondicional
   - **`goto`**
3. Repetições
   - **`for()`**
   - **`while()`**
   - **`do-while()`**

Desvio condicional

Estes comandos desviam o fluxo de execução do programa através da avaliação de uma condição ou expressão condicional.

Comando `if(), if()-else`

O comando `if()` testa o resultado de uma expressão, se o resultado for verdadeiro, o bloco associado ao `if()` é executado, se o resultado for falso e existir o bloco associado ao `else`, este é executado. Caso não exista a cláusula `else` no comando, nada é executado e a execução passa para o próximo comando.

Importante lembrar que o C considera verdadeiro qualquer valor numérico diferente de zero e falso o valor zero. Assim, são válidas também como condições, expressões com resultado numérico.

Sintaxe:

```
if(<expressão>) <comando ou bloco de comandos>
[else <comando ou bloco de comandos>]
```

Exemplos:

```
void main() {
  int a, b;
  printf("Digite dois inteiros:");
  scanf("%d %d",&a, &b);
  if(a>b) print("Maior valor: %d", a);
```

```
  else printf("São iguais ou o maior é: %d", b);
}

void main(){
  int a,b,c;
  printf("digite 3 inteiros:  ");
  scanf("%d %d %d", &a,&b,&c);
  if(b+c) printf("a/b+c)= %d", a/(b+c);
  else printf("Divisão por zero");
}
```

if´s encadeados

É muito comum na programação ser necessário encadear vários comandos `if` para testar condições mais complexas. O encadeamento pode ser feito pela parte verdadeira do comando, pela parte falsa, ou pode ser misto. No exemplo abaixo o encadeamento é feito pela parte falsa do comando, nesse caso um novo `if` é colocado após o `else` do `if` anterior.

```
if(expressão){bloco de comandos}
else if(expressão){bloco de comandos}
     else if(expressão){bloco de comandos}
          else {bloco de comandos}
```

Note que em um encadeamento como o acima, assim que uma das expressões resulta em verdadeiro o bloco de comandos correspondente é executado e nenhuma outra expressão é testada. O fluxo do programa salta todo o encadeamento e passa para o próximo comando. Abaixo, um exemplo de encadeamento pela parte verdadeira:

```
if(expressão)
  if(expressão)
    if(expressão){bloco de comandos}
    else {bloco de comandos}
  else {bloco de comandos}
else {bloco de comandos}
```

Um encadeamento pela parte verdadeira, pode ser mais confuso, principalmente se nem todos os `if´s` tiverem a parte falsa, ou seja, o `else` correspondente. Nesse caso pode não ficar muito claro a que `if` pertence cada `else`. Uma indentação correta é fortemente recomendada, pois facilita muito a visualização do encadeamento e evita erros no programa. De qualquer modo, a regra é que cada `else` está associado ao `if` mais próximo dele. No exemplo abaixo, o `else` está associado ao segundo `if`:

```
if(expressão1)
  if(expressão2) comando 1;
  else comando 2;
```

Se fosse necessário que o `else` estivesse associado com o 1º `if` então teríamos que utilizar chaves como no exemplo abaixo:

```
if(expressão1) {
    if(expressão2) comando1;
}else comando2;
```

O operador ?

O C oferece um operador que em alguns casos pode substituir o comando `if`, mas ele vai além disso e como todo operador, realiza uma operação e retorna um resultado. O operador `?` é um operador ternário, ou seja, exige 3 operandos na sua sintaxe.

Sintaxe:

```
<expressão condicional>?<expressão1>:<expressão2>
```

A expressão condicional será avaliada, se o resultado for verdadeiro, a expressão1 será resolvida e este será o resultado da operação. Se o resultado for falso, então é a expressão2 que será o resultado da operação. Veja o exemplo abaixo:

```
int x,y;
...
x = y>100?100:y;
```

Neste exemplo, `y` será comparado com `100`, se for maior que `100`, o resultado da operação será `100` e este é o valor que será atribuído a `x`, caso não seja maior que `100`, então o resultado da operação é o valor de `y`.

Note que o mesmo resultado pode ser obtido com o comando if. O código abaixo tem o mesmo efeito que o exemplo acima:

```
int x,y;
...
if (y>100) x = 100;
else x = y;
```

Apesar de o operador sempre retornar algum resultado, este pode ser ignorado se necessário, não é preciso atribuir o resultado da operação para alguma variável. Como quase todas as funções em C retornam algum valor, o operador pode ser usado para selecionar um entre dois comandos e executa-lo, como no exemplo abaixo:

```
int y;
...
y>=0?printf("positivo ou zero"):printf("negativo");
```

Note que existe um resultado já que a função `printf()` retorna um valor inteiro, no entanto, esse resultado será descartado, pois não há onde armazena-lo.

Importante: Não é possível utilizar blocos de comandos para executar mais de um comando com o operador `?`.

Comando `switch()`

Muito utilizado para menus, o comando `switch()` é capaz de testar o valor de uma variável escalar (ou seja, int ou char) com uma lista de valores possíveis. Para cada valor pode ser associado um ou mais comandos. Este comando pode substituir um encadeamento if-else-if com a vantagem de tornar o código muito mais claro e legível. No entanto o comando só é capaz de pode a igualdade de uma variável com os valores listados. É parecido com o comando `case` do pascal, mas tem algumas diferenças importantes.

Sintaxe:

```
switch(<variável>){
   case <valor1>: <lista de comandos>
                  [break;]
   case <valor2>: <lista de comandos>
                  [break;]
   case <valor3>: <lista de comandos>
                  [break;]
   .
   .
   [default : <lista de comandos>]
}
```

O comando `switch()` pode ser pensado como um único bloco de comandos com vários pontos de entrada. Este comando testa o valor da variável com os valores colocados na cláusula case. Quando uma igualdade é encontrada, os comandos passam a ser executados a partir dali, até que um comando `break` seja encontrado ou até o final do comando `switch()`. Note que pra cada valor a ser comparado há uma **lista** de comandos e não um **bloco** de comandos. Não devem ser usadas as chaves que delimitam blocos, porque elas farão com que o comando `break` não funcione corretamente. Claro que se dentro da lista de comandos houver algum comando que utiliza bloco de comandos, as chaves podem ser utilizadas sem problemas, mas nesse caso o comando `break` estará fora desse bloco. Se não for encontrado nenhum comando break, a execução continua até o final do comando `switch()`. A cláusula default é opcional e serve como ponto de entrada quando nenhum valor listado combina com o valor da variável testada.

Exemplo:

```
main(){
  int x, y;
  char opc;
  .
  .
  switch(opc){
    case 'A':
    case 'a': x = y;
              break;
    case 'B':
    case 'b': x = -y;
              break;
    default : printf("opção inválida");
}
```

Note que para o valor `´A´` não está associado nenhum comando, e também não há um comando break, então se este for o valor da variável `opc` a execução vai iniciar ali e continuar, "invadindo" os comandos associados ao valor `´a´`, até o comando `break` ser executado.

Observação: Os valores associados às cláusulas **`case`** têm que ser explícitos, ou seja, constantes. Não podem ser usadas variáveis, funções ou expressões.

Comandos de Repetição

Comando `while()`

É usado para repetir um bloco de comandos enquanto uma condição for verdadeira. O bloco de comandos será repetido até a condição se tornar falsa. Note que se a condição já inicia falsa, nenhuma repetição é feita, então pode ocorrer de nada ser executado.

Sintaxe:

```
while(<condição>)<bloco de comandos>
```

Exemplo:

```
int x = 0;
while(x<100){
  printf (”%i \n”, x );
  x++;
}
```

Comando `break`

O comando `break` colocado dentro do `while()` (ou dentro de qualquer comando de repetição) faz com que as repetições sejam abortadas imediatamente e a execução passe para o comando que estiver depois do bloco do `while()`.

Comando `continue`

Este comando dentro do comando `while()` faz com que o restante do bloco seja ignorado e a execução passa para o teste da condição. Note que este comando não aborta as repetições, apenas faz com que o restante do bloco seja ignorado e seja feito um novo teste na condição.

Exemplo:

```
int x = 0;
while (x < 100) {
  scanf (“%i“, &y);
  if (!y) continue;
  printf ( “%i “, x/y);
  x++;
}
```

Observação: Os comandos `break` e `continue` funcionam da mesma forma em todos os comandos de repetição, a saber: `while()`, `do-while` () e `for()`.

Comando `do-while()`

Bastante parecido com o comando `while()`, a diferença é que a condição é testada no final do bloco de comandos. Significa que o bloco será executado no mínimo uma vez enquanto que no comando `while()` pode ocorrer que o bloco não seja executado nenhuma vez. Os comando `continue` e `break` funcionam da mesma maneira que no `while()`.

Sintaxe:

```
do <bloco de comando>
while(<condição>);
```

Exemplo:

```
int x = 0 ;
do{
  printf(”%i \n”, x);
   x++;
} while(x<100);
```

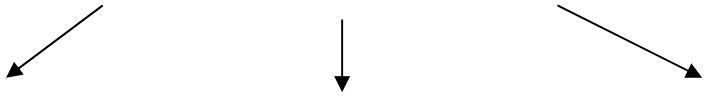
Comando `for()`

Usado para repetir um bloco de comandos, este comando é extremamente flexível e poderoso. Geralmente é utilizado com uma variável de controle, mas é de tal forma flexível que pode ser utilizado sem nenhuma variável de controle ou com várias variáveis de controle.

Quando o comando entra em execução, a expressão de inicialização é executada. Essa expressão será executada apenas uma única vez, antes de qualquer comando ou expressão. A seguir a condição é testada, se resultar em verdadeiro (ou seja, qualquer valor diferente de zero), então o bloco de comando é executado. Depois da execução do bloco, a expressão de incremento é executada e um novo teste é feito na condição, que se continuar verdadeira, o bloco será novamente executado. Esse processo continua até que a condição resulte em falso ou um comando break no interior do bloco de comandos seja executado

Sintaxe:

for ([<inicialização>] ; [<condição>] ; [<incremento>]) < bloco de comandos>



Executada apenas uma vez no início das repetições	Testada antes de cada repetição	Executado após cada repetição

Exemplo:

```
int x;
for(x=0; x<100; x++)
  printf (“%i\n“, x);
```

A expressão de inicialização e de incremento não precisam ser únicas, no caso de haver mais de uma expressão de inicialização ou incremento, devem ser separadas por vírgulas. Tanto as expressões de inicialização e incremento quanto a condição, também podem ser omitidas. No caso da condição ser omitida, o C considera que é sempre verdadeira, portanto as repetições só vão parar quando um comando `break` for executado.

Exemplo 1:
`int x,y`
`for(x=0,y=0; x+y<100; x++,y+= 5)`
`  printf (”%i %i”, x,y );`
 	
Exemplo 2:
`int x,y`
`x=y=0;`
`for(;;){`
	No exemplo 1 ao lado temos duas variáveis de controle sendo inicializadas e incrementadas no comando `for()`. Note as vírgulas separando os comandos.
	
	No exemplo 2, a inicialização foi feita antes do comando e a condição e o incremento são feitos dentro do bloco de

`printf("%i %i", x,y );`
`x++;`
`y+=5;`
`if(x+y<100) break;`
`}`
	comando, portanto todo o controle do for() foi omitido. Note que mesmo assim os ponto-e-vírgulas que separam as seções do comando precisam estar presentes. Os dois exemplos fazem exatamente a mesma coisa.

Matrizes e Vetores

Matrizes são coleções de dados do mesmo tipo que são referenciadas pelo mesmo nome. Cada elemento da coleção tem um índice associado a ele. O acesso é feito indicando o nome da matriz (coleção) seguido do índice do elemento.

Uma matriz pode ter uma ou várias dimensões. Matrizes de uma dimensão (ou unidimensionais) são também chamadas de **vetores** e funcionam como listas de dados. Nas matrizes multidimensionais, para cada dimensão é associado um índice correspondente para acessar os elementos. Assim, para acessar um elemento de uma matriz de 2 dimensões, é necessário indicar 2 índices, uma matriz de 3 dimensões exige 3 índices e assim por diante. No caso de ter 2 dimensões, a matriz pode ser pensada como um tabuleiro com linhas e colunas numeradas, onde os dois índices representam respectivamente o número da linha e o da coluna.

Seguindo o mesmo raciocínio, podemos pensar numa matriz de 3 dimensões como uma estrutura cúbica ou um paralelepípedo, que além de altura (linha) e largura (coluna) tem também profundidade. Vale notar que essas interpretações de matrizes (tabuleiros, cubos, paralelepípedos) são analogias humanas para visualizar graficamente a matriz. Na memória do computador, tudo o que existe são listas de dados indexadas. Dessa forma, matrizes com mais de 3 dimensões, embora sejam difíceis de imaginar graficamente, podem ser criadas e controladas pelos programas sem nenhum problema.

Em qualquer matriz o primeiro elemento terá sempre o índice 0 (zero) se for unidimensional ou 0,0 se for bidimensional e assim por diante.

Importante: Não há verificação de limites em matrizes no C. Por exemplo, se um vetor foi definido com 10 elementos e o programa tentar acessar o 11º elemento, não vai haver nenhum erro de compilação ou de execução, o que não significa que não há erro. Nesse caso o programa estaria invadindo a área de memória de outra variável e o resultado é imprevisível. Longe de ser um problema, essa característica do C permite implementar mecanismos de acesso a matrizes extremamente poderosos e rápidos, especialmente quando também são utilizados ponteiros. Mas o preço pago por esse poder é a responsabilidade (como sempre). O programador fica responsável por tomar os cuidados necessários para que o programa não ultrapasse o tamanho que foi definido para a matriz.

Matrizes Unidimensionais (vetores)

São listas de dados, um índice é suficiente para acessar os elementos.

Declaração:

```
<tipo de dado> <nome> [<tamanho>];
```

Exemplo:

```
int   lista [20];
float notas [40];
long int val [30];
```

Acesso

O acesso aos elementos é feito utilizando o índice do elemento entre colchetes logo após o nome do vetor. Lembrando que o primeiro elemento tem sempre o índice 0 (zero).

Exemplo:

```
Lista[0]=10;  /* Coloca o valor 10 no primeiro elemento do vetor lista.*/
Lista[1]=Lista[0];  /* Copia o valor o 1º elemento para o 2º */
```

Matriz Bidimensional

Matrizes bidimensionais têm sempre um arranjo tipo linha-coluna, exigem portanto 2 índices, um para as linhas e outro para as colunas.

Declaração :

```
<tipo de dado> <nome> [<n° de linhas>] [<n° de colunas>];
```

Exemplo :

```
 int grade [10][20]; /*Matriz de inteiros com 10 linhas e 20 colunas*/
 float Nt  [5][10];  /*Matriz de reais com 5 linhas e 10 colunas*/
```

Acesso

No acesso aos elementos, os dois índices precisam ser indicados dentro de colchetes. Note que cada índice usa um conjunto próprio de colchetes.

```
grade[0][0] = 5;
grade[0][1] = grade[0][0] + 1;
```

Matrizes de strings

Strings são por definição vetores de caracteres. O C implementa mecanismos especiais para facilitar o uso de strings, mas de fato, uma string é sempre um vetor de caracteres. Portanto, um vetor de strings, é declarado como uma matriz de caracteres, mas podemos pensar nele como um vetor onde cada elemento é também um vetor, ou no caso, a string. Veja as declarações abaixo:

```
char Nome[40];   /* string para o nome */
char ListaNomes [10][40]; /*Lista com capacidade para 10 nomes*/
char Classes [5][10][40]; /* Matriz para 5 listas de nomes */
```

Inicialização de matrizes :

Assim como os tipos simples de dados, matrizes também podem ser inicializadas no momento em que são declaradas. Para isso basta colocar entre chaves {} os valores separados por vírgulas:

```
int  valores [10]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int  grade [2][3]= {1, 2, 3,
                    2, 3, 4 };
char  Trim1 [3][10]= {"Janeiro",
                      "Fevereiro",
                      "Março"};
```

O comando #define

Como todos os comandos iniciados com o caracter #, o comando #**define** é um comando para o pré-processador e portanto, é executado antes que o programa seja compilado. O que ele faz é substituir um texto por outro no programa. O uso mais comum deste comando é para definir constantes que o programa usa, mas existem recursos avançados no comando que permitem também construir macros com parâmetros.

Na sua forma mais básica, o comando tem a seguinte sintaxe:

```
#define Nome Texto_para_substituição
```

Exemplo:

```
#define TAM 50
void main(){
  int Vetor[TAM], i;
  for (i=0; i<TAM; i++) Vetor[i] = 0;
  . . .
```

No exemplo acima, o comando `#define` fará com que em todo ponto do programa onde seja encontrado o texto "TAM", este seja substituído pelo texto "50". Isso acontece antes da compilação, portanto o compilador encontrará o programa desta forma:

```
void main(){
  int Vetor[50], i;
  for (i=0; i<50; i++) Vetor[i] = 0;
  . . .
```

Note que o comando `#define` não usa o ponto-e-vírgula como terminador. O comando termina com o final da linha. Tudo o que estiver depois da declaração do nome, é considerado texto para substituição, inclusive os espaços que houver a partir do primeiro caracter do texto. Se for utilizado um ponto-e-vírgula depois do texto 50 no exemplo acima, o ponto-e-vírgula passa a ser considerado parte do texto para substituição e será colocado também no programa, gerando erros de compilação.

O uso de definições para representar tamanhos de vetores e matrizes é quase que obrigatório em C em virtude das vantagens para o programador durante o desenvolvimento do programa e também para sua manutenção. Usando definições é possível centralizar em um único ponto do programa as dimensões dos vetores e matrizes utilizados. Fica fácil então alterar com segurança o tamanho das matrizes quando necessário. Veja o exemplo acima, é apenas um trecho de programa que inicia zerando um vetor de 50 posições. O tamanho do vetor aparece em 2 linhas do programa, na declaração do vetor e no comando for. É de se supor que o programa completo teria muito mais referências a esse valor. Se for preciso alterar o tamanho do vetor em função de algum teste ou manutenção do programa, todas as referências ao tamanho do vetor precisam ser alteradas. Utilizando definições, basta alterar a definição que tudo fica alterado automaticamente, sem o risco de esquecer alguma referência. Quando couber, os próximos exemplos utilizarão definições para as dimensões de matrizes e vetores.

Uso de comandos de repetição

Pela própria definição de matrizes, os comandos de repetição são perfeitamente adequados para percorrer ou processar matrizes. Programas que utilizam matrizes quase sempre utilizam também comandos de repetição. Todos são adequados, mas o comando **for** costuma ser mais utilizado porque tem controles que podem ser diretamente utilizados para controlar os índices das matrizes.

O exemplo abaixo utiliza um comando for para zerar um vetor de inteiros:

```
#define TAM 100
```

```
int VetInt[TAM], i;
for (i=0; i<TAM; i++) VetInt[i] = 0;
```

Para matrizes de 2 dimensões o modo mais intuitivo seria utilizar dois comandos **for** aninhados (um dentro do outro), para percorrer as linhas e colunas da matriz, como mostrado no exemplo abaixo que zera uma matriz de reais.

```
#define LINS 10
#define COLS 20
float Mat[LINS][COLS];
int lin, col;
for(lin=0; lin<LINS; lin++)
  for(col=0; col<COLS; col++) Mat[lin][col] = 0;
```

As matrizes na memória

Em C os elementos de uma matriz são sempre armazenados em posições contíguas da memória. Isso vale para elementos e também linhas, no caso de matrizes de 2 dimensões. Um elemento sempre vai ocupar na memória a posição imediatamente anterior ao próximo elemento. Da mesma forma, uma linha sempre vai estar imediatamente antes da próxima linha. O nome da matriz é na verdade um ponteiro com o endereço base da matriz, os índices servem para o C poder calcular o deslocamento a partir do endereço base. Vejamos o caso de um vetor como exemplo. O nome do vetor é o ponteiro para o endereço de memória onde inicia o vetor. O endereço de cada elemento é calculado usando o índice do elemento e o tamanho em bytes que o tipo de dado ocupa. Multiplicando um pelo outro temos o deslocamento a partir do endereço base para encontrar o elemento que vai ser acessado. A equação a abaixo descreve o modo como o endereço é calculado:

*E = base + (ind * tam)*

Onde:

E é o endereço do elemento
base é o endereço base, representado pelo nome do vetor
ind é o índice do elemento no vetor
tam é o tamanho em bytes que o tipo do elemento ocupa

No caso de matrizes com 2 dimensões, a lógica é a mesma, mas são utilizados os dois índices:

*E = base + (lin * ncols * tam) + (col * tam);*

Onde:

E é o endereço do elemento
base é o endereço base, representado pelo nome da matriz
lin e *col* são os índices da linha e coluna respectivamente
ncols é o número de colunas por linha
tam é o tamanho em bytes que o tipo do elemento ocupa

A matriz como vetor

Podemos aproveitar o fato de o C não fazer verificação de limites nas matrizes e percorrer a matriz variando apenas um índice e fixando o outro, dessa forma apenas um comando for é necessário para zerar a matriz. No exemplo abaixo, a matriz tem 200 elementos organizados em 10 linhas e 20 colunas. Fixando a linha 0, temos então as colunas de 0 a 19. Mas como o C não faz verificação de limites, nada impede que acessemos a coluna 20, 21, 22, etc. O C simplesmente vai usar o índice fornecido para calcular o endereço na memória do elemento desejado. Veja que aplicando a equação para os valores (1,0) que corresponde ao 1º elemento da segunda linha e para os valores (0,20) que indicaria o 21º elemento da primeira linha, obtemos

exatamente o mesmo valor de endereço. Portanto, podemos então percorrer a matriz apenas com um índice, como se fosse um vetor, como mostrado no exemplo abaixo. Note que sendo uma matriz de duas dimensões, os dois índices são necessários, mas o índice das linhas fica fixo no zero, enquanto o das colunas varia. A definição NCEL contém o número de células na matriz.

```
#define LINS 10
#define COLS 20
#define NCEL LINS*COLS
float Mat[LINS][COLS];
int i;
for(i=0; i<NCEL; i++)
  Mat[0][i] = 0;
```

Busca seqüencial em vetores

Dados precisam ser acessados, do contrário são inúteis, mas antes de serem acessados, eles têm que ser localizados e para isso em geral é necessário fazer uma pesquisa na base de dados.

O método de pesquisa está diretamente relacionado com a maneira com que os dados estão organizados. Se não existe nenhuma ordem nos dados, a única forma de pesquisa possível é a busca seqüencial. Este método consiste em "olhar" os dados um por um seqüencialmente, até encontrar o dado procurado ou até chegar ao fim da base de dados (ou vetor, no caso).

Ordenação de vetores

Na computação, talvez não existam outras tarefas que sejam tão importantes ou tão intensamente pesquisadas quanto as tarefas de ordenação e pesquisa (busca). Quase todos os sistemas de algum porte incluem rotinas de ordenação e pesquisa. Computadores são especialmente úteis para processar grandes quantidades de dados e estes dados precisam ser acessados de preferência rapidamente. Isto significa que é preciso encontrar um dado entre centenas, milhares e as vezes milhões, no menor tempo possível. Os dados são muito mais rapidamente encontrados quando estão ordenados. Usemos como exemplo uma lista telefônica. É fácil encontrar um nome na lista porque ela está sempre ordenada em ordem alfabética. Agora imagine se não houvesse nenhuma ordem nos nomes da lista. O tempo necessário para encontrar um nome aumentaria violentamente, isto também é verdade na computação.

Existem basicamente três categorias de algoritmos de ordenação, descritos a seguir:

- por troca
- por seleção
- por inserção

Por troca : Esta categoria de algoritmos troca os elementos de posição, até que todos estejam ordenados. Para entender este tipo de algoritmo, imagine que tenhamos um naipe completo de um baralho para ordenar.

Para começar, enfileiramos as cartas sobre a mesa, e então procederíamos trocando as cartas de posição duas a duas, até que todas estejam ordenadas. O algoritmo desta categoria mais conhecido (e também o mais lento) é o algoritmo da bolha. Este algoritmo compara elementos adjacentes e quando necessário os troca de posição. O processo é repetido até esgotar todas as possibilidades de troca. Por ser um método muito lento e obsoleto, não nos aprofundaremos nele.

Por seleção : A idéia básica deste tipo de algoritmo é selecionar o menor elemento do vetor (para ordenação crescente) e trocá-lo de posição com o primeiro elemento. A seguir seleciona-se o menor elemento entre os restantes para que seja trocado com o segundo elemento. O processo é repetido até

que todas as posições tenham sido testadas. No exemplo do baralho, colocaríamos as cartas sobre a mesa e procuraríamos a de menor valor para trocar de posição com a primeira carta. Em seguida procuramos a que tem o segundo menor valor e a trocamos com a segunda carta. Quando completássemos todas as posições, o baralho estaria ordenado.

O algoritmo por seleção mais popular é o algoritmo do chocalho. Tem este nome porque os elementos vão e voltam pelo vetor como se estivessem chacoalhando. Embora seja em média um pouco mais rápido do que o da bolha, este algoritmo ainda é um dos mais lentos que existem, mas sobrevive graças à sua simplicidade sendo muito empregado, especialmente para vetores pequenos com até algumas centenas de elementos.

Por inserção : Este método de ordenação, inicialmente ordena os dois primeiros elementos e a seguir insere os outros na ordem correta. No exemplo do baralho, seguramos duas cartas e as ordenamos, a seguir pegamos a terceira carta e a colocamos na sua posição ordenada em relação às outras duas. O processo continua até acabarem as cartas, quando então o baralho estará ordenado.

Este método é extremamente mais rápido do que o método da bolha ou do chocalho citados acima, embora seja muito mais trabalhoso e complexo na sua implementação. Há muitas maneiras de implementar este algoritmo, sendo que as mais eficientes utilizam a busca binária para determinar a posição em que cada novo elemento será inserido no vetor.

O programa abaixo implementa um algoritmo de ordenação por inserção mais simplificado que aproveita a idéia do método da bolha. Não é um método dos mais eficientes, mas ainda assim é bem mais rápido que os algoritmos da bolha e do chocalho e também é simples de implementar. O algoritmo funciona da seguinte maneira: os dois primeiros elementos do vetor são ordenados. Depois, do 3º até o último elemento são feitas comparações com os elementos vizinhos anteriores, trocando as posições quando necessário. Ao completar o processo com o último elemento, o vetor estará ordenado

```
#define TAM 500
int vet[TAM], t, i, ind;
. . .
for(i=1; i<TAM; i++) { /* percorre do 2o ao ultimo elemento */
  ind = i;
  while(vet[ind]<vet[ind-1]) {
    /* Troca as posições ind e ind-1 */
    t=vet[ind];
    vet[ind]=vet[ind-1];
    vet[ind-1]=t;
    /* Decrementa ind */
    ind--;
    /* testa final do vetor */
    if(!ind) break;
  }
}
```

Criando funções e bibliotecas

Função é uma unidade autônoma de código do programa geralmente desenhada para cumprir uma tarefa em particular. Programas em C consistem em várias pequenas funções. Podem receber parâmetros que devem ser separados por vírgulas. Podemos pensar numa função como um bloco de código que tem um nome (identificador) e pode receber valores (parâmetros) para uso do código. Durante o curso temos utilizado funções que estão implementadas nas bibliotecas padrão da linguagem. Por exemplo **`scanf()`** e **`printf()`** são funções implementadas na biblioteca **`stdio.h`**. Também já criamos pelo menos uma função nos programas, que é a função **`main()`**,

presente em todos os programas em C. Criar outras funções não é muito diferente de criar a função `main()`, a não ser pelo fato da função `main()` ter restrições que não existem nas outras funções. Toda função em C segue a seguinte sintaxe:

```
<tipo> <nome> ([parâmetros]) {
   <declaração de variáveis>
   <bloco de comandos>
   [return <valor>]
}
```

O tipo pode ser qualquer tipo válido na linguagem e indica o tipo de dado que a função retorna. Caso a função não retorne nenhum valor (caso das procedures do Pascal) o tipo deve ser **`void`**.

O nome é o identificador da função, segue as mesmas regras dos identificadores das variáveis. É pelo nome que a função será chamada pelo programa.

Os parâmetros são as informações que a função recebe para poder executar sua tarefa ou retorna quando a tarefa está concluída. Se houver mais de um devem ser separados por vírgulas. Mesmo que não exista nenhum parâmetro, os parêntesis que os envolvem são sempre obrigatórios.

A declaração das variáveis locais da função deve ser feita sempre antes de qualquer comando. A seguir vem o bloco de comandos. Caso a função retorne algum valor, deve usar o comando **`return`** para indicar o valor de retorno. O comando **`return`** também serve para encerrar a execução da função.

Tipo da função

O tipo da função especifica que tipo de valor será retornado pela função. O tipo **`void`** indica que a função não tem retorno. A função pode retornar qualquer tipo e valor válido no C, inclusive estruturas como registros (**struct**) e ponteiros, geralmente usados para retornar matrizes e vetores.

Parâmetros formais

As funções podem receber valores para executar sua tarefa. As variáveis que recebem estes valores são chamadas de **parâmetros formais** da função. Há dois tipos de parâmetros, os usados nas **chamadas por valor** e nas **chamadas por referência**.

Chamada por valor

Nesse caso os valores passados à função são copiados para os parâmetros formais. Se forem usadas variáveis na chamada da função, possíveis alterações nos parâmetros ocorridas no corpo da função não alteram as variáveis.

Chamada por referência

Quando os parâmetros são passados por referência, o que a função recebe não é apenas um valor que a função precisa, mas uma referência a uma variável, ou em outras palavras, um ponteiro para uma variável. Através do ponteiro, a função pode não apenas ler o conteúdo da variável, mas também alterá-lo. Parâmetros passados por referência são muito úteis para modificar o conteúdo de variáveis locais no ponto de chamada, e também para a função retornar valores quando apenas o retorno normal da função (pelo comando **`return`**) não é suficiente. Mais adiante este assunto será visto com mais detalhes, após o estudo de ponteiros.

Exemplo (chamada por valor):

```
int soma (int a, int b) {
   int res;
   res = a+b;
   return res;
}
```

No exemplo acima foi criada uma função chamada **soma** que recebe dois valores inteiros como parâmetros (**a** e **b**) e retorna a soma dos dois valores. Veja que foi declarada uma variável local (**res**) para receber o resultado da soma. No final o conteúdo dessa variável é retornado com o comando **return**.

Protótipos de funções

Protótipo de função é uma declaração feita para o compilador saber que determinada função existe, antes que ela seja efetivamente compilada. Por exemplo, sem o uso de protótipos a função Soma teria que ser implementada (e portanto compilada) antes de qualquer função que a chame (inclusive a função **main**), caso contrário o compilador acusaria um erro nas chamadas à função **soma** que fossem compiladas antes dela. O protótipo permite informar ao compilador que determinada função existe e evitar esse erro de compilação. A forma completa do protótipo é a seguinte:

```
<tipo > <nome > (<lista de parâmetros>);
```

Importante o ponto-e-vírgula no final encerrando a declaração. A partir do momento que o protótipo for compilado, o compilador sabe como tratar as chamadas à função, mesmo antes de compilar efetivamente a função.

Em geral os protótipos das funções são colocados no topo do arquivo para que todas as funções implementadas no arquivo tenham acesso às funções prototipadas, independente da ordem em que foram implementadas no arquivo.

Os protótipos também servem para dar acesso a funções em bibliotecas. Um arquivo de biblioteca pode ter uma quantidade enorme de funções, mas nem todas elas foram feitas para uso do programador. Quando incluímos um arquivo com extensão H (header-cabeçalho) em um programa, estamos incluindo na verdade um arquivo que tem principalmente protótipos das funções que estão disponíveis para determinada biblioteca. Estes arquivos podem conter também definições e variáveis globais, em geral relacionados com as funções prototipadas nele. O arquivo header não contém a implementação de nenhuma função, as implementações estão em outro arquivo, ou podem estar distribuídas entre vários outros arquivos que são incluídos no arquivo header.

Bibliotecas de funções

Biblioteca é um conjunto de funções que ficam disponíveis para uso do programador. Todas as funções do C estão em alguma biblioteca, por esse motivo, desde o início do curso foi necessária a inclusão de bibliotecas nos programas. Entretanto, o programador pode criar (e normalmente cria) suas próprias bibliotecas com funções de uso mais comum. A maneira mais simples de criar uma biblioteca é criar um arquivo com extensão C, como um arquivo de programa, só que sem nenhuma função main, e depois incluí-lo no programa usando o comando **#include**.

Também é possível usar um arquivo header para prototipar as funções disponíveis, isso e necessário se a biblioteca começar a ficar maior do que algumas funções. Nesse caso o arquivo header deve ter um comando **#include** para o arquivo C onde as funções estão implementadas. Ao usar o include, utilize aspas (“ “) no lugar dos sinais < > ao especificar os arquivos para fazer com que o compilador procure o arquivo no diretório de trabalho atual (onde o programa está sendo compilado).

Exemplo:

Arquivo header `minhabib.h:`

```
#include "minhabib.c"
int soma( int a, int b );
```

Arquivo C `minhabib.c:`

```
int soma( int a, int b )
{
  return a+b;
}
```

Uso de Ponteiros

O entendimento e uso correto de ponteiros na linguagem C é importantíssimo para o desenvolvimento de programas eficientes. Quase todos os programas em C utilizam largamente os ponteiros. Na verdade, mesmo sem saber, qualquer programador que utilize uma matriz ou vetor no programa, está utilizando ponteiros. Mas ponteiros não servem apenas para manipular matrizes, embora esta seja uma de suas funções mais importantes. Também podem ser usados para alocar e liberar blocos de memória dinamicamente, para alterar os parâmetros das funções e além disso fornecem maneiras alternativas de acessar as variáveis do programa. Todos esses recursos trazer muito poder ao programador e por isso eles são largamente usados nos programas, mas todo esse poder tem um custo, é muito fácil criar erros perigosos e difíceis de detectar com o uso incorreto de ponteiros. Por esta razão, todo cuidado é pouco ao utilizá-los.

Ponteiros são variáveis capazes de armazenar endereços de memória. Podemos pensar na memória do computador como um enorme vetor de bytes, onde cada byte tem seu próprio endereço (ou índice). Estes endereços podem ser armazenados em ponteiros. Além disso, os ponteiros também podem ter tipos associados, que informam que tipo de dado é encontrado no endereço indicado pelo ponteiro. Também existem ponteiros sem tipo (**void**) que podem ser usados para alocar blocos de memória sem formatação nenhuma.

Declaração:

Utiliza-se o caracter * para indicar que uma variável será um ponteiro:

```
<tipo> *<nome>;
```

Exemplo:

```
int *ptInt, i;
```

No exemplo acima foi declarado um ponteiro do tipo `int` chamado `ptInt` e uma variável inteira chamada `i`.

Acesso

Ao usar os ponteiros é importante diferenciar o endereço armazenado no ponteiro e o dado armazenado no endereço. O ponteiro permite o acesso às duas informações e por este motivo sua utilização é um tanto confusa no início. Lembrando: o ponteiro armazena um endereço e neste endereço está algum dado. É possível acessar ou modificar o endereço armazenado no ponteiro e

também é possível acessar ou modificar o dado armazenado no endereço. Escrevendo apenas o nome do ponteiro, o acesso é ao endereço armazenado nele.

Exemplo:

```
i = 10;
ptInt = &i;
```

Neste exemplo, foi utilizado o operador & para retornar o endereço da variável `i`, que está sendo atribuído a `ptInt`. A partir do trecho acima, o ponteiro `ptInt` contém o endereço da variável `i`.

Acesso ao dado armazenado no endereço do ponteiro

Para acessar o conteúdo da posição de memória é preciso utilizar o caráter * antes do nome do ponteiro. Este caracter indica que o que está sendo acessado não é o endereço, mas o conteúdo desse endereço.

Exemplo:

```
int *ptInt, i;
(1) i = 10;
(2) ptInt = &i;
(3) *ptInt = 20;
(4) printf("i = %d", i);
```

No exemplo acima a linha (1) atribui o valor `10` para a variável inteira `i`. Na linha (2) o ponteiro `ptInt` recebe o endereço da variável `i`. Na linha (3) está sendo atribuído o valor `20` ao endereço de memória apontado por `ptInt`, como é o endereço da variável `i`, o conteúdo dela mudou para `20`. Ao imprimir o valor de `i` na linha (4), o que aparece na tela é o valor `20`.

Importante: Nunca utilize um ponteiro que não está inicializado. Antes de acessar o endereço de memória apontado pelo ponteiro, tenha certeza que é um endereço válido. Poucas coisas trazem mais problemas para o programador do que ponteiros não inicializados ou com valores incorretos, especialmente quando é necessário escrever no endereço, como no exemplo acima. Antes de receber qualquer valor, um ponteiro normalmente é criado com o valor nulo, ou seja, zero. Escrever no endereço de memória zero provocará uma queda do sistema todo, provavelmente reinicializando o computador.

Valor nulo para ponteiros

O C define a constante **`NULL`** como o valor nulo para ponteiros. Frequentemente funções que retornam ponteiros costumam usar o **`NULL`** para indicar que a operação não teve sucesso. Após cada operação envolvendo ponteiros é uma boa idéia verificar se o valor resultante é diferente de **`NULL.`**

Aritmética de ponteiros

As únicas operações aritméticas válidas para um ponteiro são a soma e a subtração com inteiros e por conseqüência, o incremento e decremento. Entretanto quando um ponteiro é incrementado ou decrementado, o que ocorre é que o incremento ou decremento será feito considerando como unidade a quantidade de bytes que o tipo do dado apontado pelo ponteiro ocupa. Por exemplo, considere o trecho de programa abaixo:

```
int *pti, vti[10];
(1) pti = &vti[0];
(2) *pti = 0;
```

```
(3) pti++;
(4) *pti = 0;
```

`pti` é um ponteiro para inteiros e `vti` um vetor de inteiros. Na linha (1) `pti` recebe o endereço do primeiro elemento do vetor, portanto na linha (2) o primeiro elemento do vetor recebe o valor 0. A linha (3) incrementa o endereço armazenado em `pti`, mas incrementa, não 1, mas 2 bytes, porque o valor inteiro no Turbo C ocupa 2 bytes, o que faz o ponteiro `pti` apontar para o próximo valor inteiro. Por isso na linha (4) o segundo elemento do vetor recebe o valor 0.

Ponteiros e matrizes

Outra forma de fazer a mesma coisa do que no exemplo anterior, mas sem modificar o endereço armazenado em `pti` é usar a seguinte notação:

```
*(pti+1) = 0;
```

Neste caso estamos adicionando os dois bytes de um inteiro ao endereço armazenado em `pti` apenas para a atribuição, mas o endereço em `pti` não é modificado. Esta notação produz exatamente o mesmo efeito que a notação abaixo:

```
pti[1] = 0;
```

Os colchetes colocados após um ponteiro indicam que deve ser somada uma quantidade de bytes ao endereço armazenado nele antes de realizar a operação indicada, neste caso, uma atribuição. Note que esta notação é exatamente a mesma usada para acessar elementos de vetores e matrizes, isso ocorre porque os identificadores de vetores e matrizes são na verdade ponteiros e os índices colocados entre os colchetes indicam o deslocamento a partir do endereço base, que é o endereço do primeiro elemento do vetor ou matriz.

Quando um vetor é declarado no programa o que acontece é que o C cria um ponteiro do tipo indicado pelo vetor, aloca memória suficiente para conter todos os elementos do vetor e depois coloca o endereço inicial da memória alocada no ponteiro criado para esse fim. Tudo isso é feito automaticamente, apenas declarando o vetor, por isso é possível tratar um ponteiro como se fosse um vetor, utilizando índices entre colchetes.

Muito importante: Apesar de o identificador de um vetor ser também um ponteiro, se for declarado como vetor, o endereço base nunca deve ser modificado. Existe o risco de no mínimo perder o acesso ao vetor, mas esse não é o pior problema. Por exemplo, se o vetor foi declarado em uma função, quando a execução sair da função, o C vai tentar liberar a memória que foi alocada para o vetor. Se o endereço foi alterado, não tem como o C liberar essa memória corretamente e algo muito ruim pode acontecer com o programa, já que vai ser liberada a porção errada de memória.

Chamadas por referência em funções

Em algumas situações é preciso que uma função possa alterar o conteúdo das variáveis usadas nos argumentos que são enviados à função. Tomemos como exemplo a função **`scanf()`**. Esta função foi escrita para ler variáveis do teclado ou do dispositivo que estiver associado ao **`stdin`** (stream de entrada padrão do C). Note que ao ler variáveis do teclado, não importa à função **`scanf()`** qual é o conteúdo da variável, porque este conteúdo será perdido depois da leitura, portanto o **valor** da variável não importa, o que a função realmente precisa é de uma **referência** à variável, uma forma de poder alterar o conteúdo dela. No C, esta forma é sempre através de ponteiros. Portanto para que a função **`scanf()`** possa alterar o conteúdo de uma variável ela precisa receber um ponteiro para esta variável. É por este motivo que sempre utilizamos o operador **&** na frente de variáveis numéricas ao

ler estas variáveis com a função. O operador **&** retorna o endereço da variável que será lida, ou seja, um ponteiro, e com esse endereço a função consegue alterar o conteúdo da variável depois da leitura.

Para receber parâmetros por referência a função precisa declarar ponteiros na lista de parâmetros e portanto, as chamadas à função precisam enviar ponteiros nos argumentos. O exemplo abaixo implementa uma função que troca os valores de duas variáveis inteiras enviadas à função:

```
void swap( int *a, int *b)
{  int tmp;
   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Como a função declara ponteiros nos parâmetros, é necessário enviar ponteiros nos argumentos. Se no ponto de chamada foram declaradas variáveis inteiras, então é necessário utilizar o operador de endereço (&) para chamar a função, como no exemplo abaixo:

```
void main()
{ int x, y;
  x = 10;
  y = 20;
  swap(&x, &y);
  printf( "x = %d   y=%d", x, y);
}
```

Desta forma a função `swap()` pode acessar as variáveis `x` e `y` e trocar seu conteúdo, como é o objetivo da função.