Introdução à linguagem C

Centro Nacional de Alto Desempenho

CENAPAD-SP

Prof. Jorge L. Díaz Calle

Novembro de 1998

Centro Nacional de Alto Desempenho em São Paulo
CENAPAD-SP.

Noviembre de 1998.

Introdução

Este curso visa ensinar ao aluno os conceitos básicos da linguagem de programação C, cujas virtudes mais importantes são a sua versatilidade, confiabilidade, regularidade e fácil uso (é uma linguagem amigável). Uma das grandes vantagens do C é que ele possui tanto características das linguagens de programação de "alto nível" quanto de "baixo nível", isto é, a linguagem C é um software voltado para o desenvolvimento de programas robustos e eficientes. Para aprendê-lo não é necessário o conhecimento de nenhuma outra linguagem de programação prévia, embora facilite a aprendizagem uma boa familiaridade com computadores.

A genealogia do C é simples. O primeiro ancestral do C é a linguagem Algol60, desenvolvida por um Comitê Internacional em 1960. O Algol60 apareceu poucos anos após o Fortran, embora seja muito mais sofisticado do que o Fortran. Apesar de suas virtudes, como regularidade da sintaxe e a sua estrutura modular, foi considerado abstrato e geral demais. Em 1963, entre Cambridge e a Universidade de Londres, foi criado o CPL, ou Linguagem de Programação Combinada, o que foi a primeira tentativa de trazer o Algol60 à terra. Continuou grande e complexo. O BCPL, Linguagem básico de programação combinada, tentou resolver o problema levando ao CPL as suas características básicas. Seu inventor foi Martin Richards, em Cambridge, no ano de 1967. Em 1970, nos Laboratórios Bell, Ken Thompson derivou a linguagem B, mais uma simplificação do CPL.

Nos mesmos Laboratórios da Companhia Telefônica Bell, Dennis Ritchie, em 1972, implementou o C pela primeira vez rodando o sistema operacional UNIX. O sucesso de Ritchie com o C foi baseado na recuperação da generalidade perdida, principalmente utilizando habilmente os tipos de dados e sem sacrificar a simplicidade grandemente procurada pelo BCPL e o B.

O C tem a coerência das linguagens de programação pensadas por uma única pessoa, como o BCPL, o B, o Lisp, o Pascal etc. Foi formada uma organização para elaborar e manter um padrão do C, é o ANSI C. Neste curso, estuda-se principalmente os comandos desta padronização chamada ANSI C ou C padronizado pela ANSI.

Na primeira aula, faz-se entrega de uma coleção de exemplos, já digitados, para que o aluno analise, compile e faça rodar. Sugere-se que o aluno realmente trabalhe com todos estes exemplos, o que lhe permitirá ganhar maior experiência de programação. O aluno pode modificar à vontade o código exemplo fornecido.

Fazendo isto, é provável que ele gere programas executáveis com um outro compilador, achando outros problemas particulares da sua máquina, e estas dúvidas devem ser apresentadas na próxima aula. Isto não significa que o aluno deve reduzir o seu estudo aos exemplos dados. Para aprender a programar em C, além do domínio da linguagem em si, é necessária familiaridade com o compilador e experiência em achar "bugs" nos programas.

Isto é, o conhecimento do C ou de uma outra linguagem de programação transcende o conhecimento de estruturas e funções. Então, é importante que o aluno digite, compile e execute os programas decorrentes dos exercícios que serão apresentados como trabalhos para fora das aulas.

Se o aluno tem um computador a sua disposição e não tem um compilador C, observe o seguinte:

1. No caso de máquinas Unix, elas têm junto o compilador cc. Às vezes, este não é padrão ANSI C, então aconselha-se utilizar o compilador gcc da GNU.
2. Caso seja DOS ou Windows, as melhores opções são o ambiente Borland ou o Visual C.
3. Existe um catálogo de compiladores de domínio público, http://www.idiom.com/free-compilers/

A primeira experiência e a primeira compilação

Apresenta-se nesta seção o primeiro programa exemplo, que é muito simples, e permitirá conhecer a biblioteca padrão para entrada e saída de dados, a idéia inicial do formato de um programa em C e permitirá efetuar a nossa primeira compilação. Leia bem, eu disse primeira compilação e não primeira complicação.

Inicia-se esta seção, indicando que o C é **caso sensitivo**, isto é, as maiúsculas e minúsculas são diferenciadas. Por exemplo, uma variável chamada *contador* é diferente de uma outra chamada *Contador*, e também de *contadoR*, *CONTAdor*, etc.

O exemplo está no arquivo ex01.c. Abra-o com o editor de texto da sua preferência, leia, pergunte e compile. Antes de responder as perguntas sobre a compilação, deve-se entender o texto do arquivo exemplo. O objetivo do programa exemplo é mostrar na tela uma mensagem de boas vindas. O arquivo foi gerado por um editor de texto comum, e nele está-se tentando dizer ao computador que mostre a mensagem. A linguagem utilizada para transmitir as instruções é a linguagem C, que é acessível aos humanos, mas não é acessível diretamente para o computador. O computador apenas executa as instruções que tem no seu repertório a nível de máquina, isto é, as instruções que utilizam os programadores a nível da linguagem assembler. Para utilizar C é necessário um programa tradutor de instruções em C nas suas equivalentes a nível máquina. Estes programas tradutores são chamados de compiladores.

O escrito nos arquivos exemplos utilizando a linguagem C são chamados de códigos fonte. Então a tarefa do compilador é traduzir o código fonte em instruções que o computador possa entender e executar. O produto gerado pelo compilador é um arquivo com o chamado código executável, isto é o mesmo programa original na forma que o computador possa ler e executar.

Um programador em C não apenas utiliza a sintaxe da linguagem para gerar um código fonte, ele também utiliza abreviaturas para simplificar o código. Estas abreviaturas não são conhecidas pelo compilador, então devem ser expandidas antes de passar o código para o compilador. Existe em C, um pré-processador C, que toma o código fonte de um programa e gera um outro código fonte expandido, deixando-o pronto para ser traduzido pelo compilador.

Por outro lado, quando se implementa um programa visando algum objetivo especifico, geralmente utilizam-se vários arquivos de código, e então é necessário ligar todos eles. Esta tarefa é do link-editor, ele ligará todo o código necessário gerando apenas um arquivo executável. Este arquivo executável é chamado **a.out** quando nada é especificado.

Felizmente, tudo isto é escondido para os simples mortais, e todos os passos mencionados, após ter editado o código fonte, são executados no Unix simplesmente digitando uma das seguintes linhas de comandos:

1. **cc ex01.c**
2. **cc ex01.c –o nomeexecutavel**
3. **cc -o executavel -lm ex01.c ...(outros adicionais)**
4. **xlc -o executavel ex01.c**

A primeira linha de comando compila o código fonte ex01.c e gera um executável chamado a.out. Na Segunda e na terceira linha a opção –o permite atribuir um nome ao arquivo executável, e na terceira linha –lm força ao link-editor a considerar a biblioteca pronta do C com funções matemáticas especiais.

O programa executável pode ou não realizar o que o programador projetou. Quando não tenha sucesso, procure o erro no código fonte com um editor de texto, pois é provável que exista um erro lógico no programa. Após, salve as mudanças, compile e execute novamente.

Editor de texto :	**Código fonte em C**
Compilando :	*Pré-processador C*
	Código fonte expandido em C
	Compilador C
	Código em assembler
	Montador
	Código objeto do programa e
	Biblioteca de arquivos
	Link-editor
	Código executável
Executar :	**Nome do executável.**

E o código exemplo ex01.c? Abra-o e analise-o. Observe o formato utilizado e as partes que podem ser identificadas nesse formato. Como mencionado acima, existem várias bibliotecas prontas para facilitar a entrada e saída de dados ou informação de e para o computador. Na linguagem C, existe a biblioteca padrão de entrada/saída, que é incluída simplesmente digitando a expressão **#include <stdio.h>**. Esta biblioteca é um conjunto de funções que permitem a interface com o usuário, isto é, neste arquivo existem definições de funções úteis para entrada e saída padronizada de dados. Toda vez que se queira usar uma destas funções deve-se incluir este comando. O C possui diversos arquivos-cabeçalhos.
Por exemplo, em stdio.h foi definida a função **printf**(...) para dizer ao computador o que é e como ele deve mostrar na tela alguma informação. A função printf( ) mostra na tela a

cadeia de caracteres (string) “Alo pessoal” que é passada como argumento. O \n é uma constante chamada de constante barra invertida. O \n é de "new line" e ele é interpretado como um comando de mudança de linha, isto é, após imprimir a string, o cursor passará para a próxima linha.

```
/*Comentários referentes ao programa */
#include <stdio.h>

main( ) {
        printf("Alo pessoal. \n");
}
```

Compile o exemplo ex01.c. Analise o código e verifique se o programa faz o que você espera. Antes de iniciar o estudo da sintaxe da linguagem C estabelecem-se algumas convenções :

1. **expressão** : é a representação de uma ou várias ações qualquer. Exemplo : soma = a + b
2. **instrução ou sentença** : é uma expressão terminada em ponto e vírgula. É a menor unidade independente em C. Exemplo : soma = a + b;
3. **função** : é um conjunto de instruções logicamente encadeadas.
4. **programa** : é um conjunto de funções, contendo uma função (principal) chamada **main**( ). O programa ao ser executado começa com a primeira instrução dentro da função main().

No primeiro exemplo, ex01.c, observam-se as seguintes partes:

1. Linhas de comentários, que são necessárias a fim de explicitar o objetivo do código, ajudando a elucidar o funcionamento do mesmo. Os comentários são dados utilizando /* e */ . O compilador C desconsidera qualquer coisa que esteja começando com /* e terminando com */, mesmo tendo várias linhas.
2. Diretivas para o pré-processador. Elas iniciam-se com o símbolo #. No exemplo, #include <stdio.h>, indica ao pré-processador que deve incluir o código (pronto) do arquivo de cabeçalho stdio.h no lugar dessa linha. Então, o resultado de se incluir um arquivo cabeçalho é o mesmo que se fosse incluso o texto do arquivo naquela posição. Isto economiza espaço.
3. A linha main( ) define uma função de nome main. Todo programa em C deve ter uma função main, pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves **{ }**. O código que estiver dentro das chaves será executado seqüencialmente quando a função for chamada.

Ao compilar os primeiros exemplos, algum compilador C pode dar mensagens de aviso do tipo **warning**, que não impedem o sucesso da compilação. Nos exemplos dados pode acontecer isto porque, por default, toda função em C retorna um inteiro. Quando não é retornado este inteiro, o compilador pode mandar uma mensagem do tipo "Function should return a value.". Por enquanto, esqueça estas mensagens, mais tarde aprenderá como escrever funções direito.

Fundamentos da linguagem C

Estuda-se aqui a **sintaxe** da linguagem C, a sintaxe é o conjunto de regras detalhadas para cada construção válida na linguagem C.

Variável e constante

Os dados ou valores a manipular em um programa podem ser variáveis ou constantes. Em C uma **constante** é um espaço de memória cujo valor não deve ser alterado durante a execução de um programa. E uma **variável** é um espaço de memória que recebeu um nome e armazena um valor que pode ser modificado.

Os nomes utilizados para referenciar variáveis, funções ou vários outros objetos definidos pelo usuário são chamados de **identificadores**. Os primeiros 32 caracteres são significativos, diferenciando-se as maiúsculas das minúsculas. Os identificadores devem satisfazer duas condições: começar com uma letra ou sublinhado (_), e os caracteres subseqüentes devem ser letras, números ou sublinhado (_). O identificador de uma variável tem mais duas restrições, não pode ser igual a uma palavra reservada da linguagem C (palavra-chave), nem igual ao nome de uma função declarada pelo programador ou pelas bibliotecas do C. As palavras-chave são identificadores predefinidos que possuem significados especiais para o compilador.

Palavras-chave

asm	const	else	for	Near	sizeof	union
auto	continue	enum	goto	Register	static	unsigned
break	default	extern	if	Return	struct	void
case	do	far	int	Short	switch	volatile
char	double	float	long	Signed	typedef	while

Os **tipos de dados** definem as propriedades dos dados manipulados em um programa. Quando você declara um identificador dá a ele um tipo de dado. Um tipo de dado determina como o valor desse dado será representado, que valores pode expressar e as operações que podem ser executadas com estes valores.

Todas as variáveis e as constantes possuem uma característica comum, um tipo de dado associado. Não é necessário especificar o tipo de dado de uma constante pois ele é determinado pelo seu valor. Entretanto, ao declarar um variável, além de escolher um nome apropriado para ela, deve-se dizer ao compilador que tipo de informação deseja-se armazenar nela.

Tipos de dados

Tipo de dado		Armazenamento	Intervalo de valores	Observações
char		1 byte	-128 a 127	Pelo menos 8 bits
int		2 bytes	-32 768 a 32 767	Pelo menos 16 bits
long int		4 bytes	-2 147 483 648 a 2 147 483 647	O dobro de um inteiro
unsigned ...		idem	0 a 2 * medida	Sem sinal
float		4 bytes	3.4e-38 a 3.4e+38	Simples precisão
double		8 bytes	1.7e-308 a 1.7e+308	Doble precisão
pointer		2 (4) bytes	ponteiro – perto (longe)	

Às vezes em um compilador pode-se encontrar uma faixa maior do que a mostrada na tabela, mas não uma faixa menor.

O C tem cinco tipos básicos: char, int, float, double, void. O char é um tipo de dado numérico, mas é associado com o conjunto de caracteres ASCII(como as letras do alfabeto). O int é para armazenar valores numéricos inteiros. O float e o double são para armazenamento de valores numéricos em dígitos de precisão. O float em precisão simples e o double em dupla precisão. O void (vazio em inglês) é um tipo especial, e o seu estudo será feito posteriormente.

Exceto o void, os tipos de dados básicos podem ter vários modificadores. Um modificador é utilizado para alterar o significado de um tipo básico para adaptá-lo às necessidades da situação. Os modificadores de tipo do C são quatro: signed, unsigned, long e short. Os quatro podem ser aplicados a inteiros e caracteres. Ao float não se pode aplicar nenhum e ao double pode-se aplicar apenas o long. A intenção é que short e long devam prover tamanhos diferentes de inteiros onde isto for prático.

Uma variável de um tipo pode ser convertida para um outro tipo utilizando o **conversor de tipos** ou **cast**. Porém não são todos os tipos que podem ser convertidos para um outro tipo com segurança. Deve-se ter cuidado com o tamanho de cada tipo de dado.

Exemplo,

```
int segundos;
double minutos = ((double)segundos)/60.;
```

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

tipo_da_variável *lista_de_variáveis***;**

As variáveis na lista de variáveis terão todas o mesmo tipo e deverão ser separadas por vírgula. Como o tipo default do C é o int(inteiro), ao declarar variáveis int com algum dos modificadores de tipo, basta colocar o nome do modificador de tipo. Assim para um long int basta declarar long, para um unsigned int basta declarar unsigned. Declarar signed para int é redundante.

Exemplo :

```
int  alunos, cursos, iterações;
long  fatorial;
double  custo, valor;
```

Pode-se inicializar variáveis no momento de sua declaração com a seguinte forma geral :
tipo_da_variável *nome_da_variável* = **constante**;
Isto é importante pois quando é criada uma variável com o C, ela não é inicializada, isto é, até que um primeiro valor seja atribuído à nova variável, ela tem um valor indefinido e que não pode ser utilizado para nada. Nunca presuma que uma variável declarada vale zero ou qualquer outro valor.

Exemplos de inicialização :

```
char  letra = 'D';
int  cursos = 3;
```

Constantes em C

Uma **constante literal** é a variedade mais comum. Elas compõem-se de valores como 3.14, 2, ou ainda de informações de string digitados diretamente no texto do programa, “Alo pessoal.”. O C também permite inserir constantes numéricas hexadecimais(base dezesseis) ou octais(base oito). As constantes hexadecimais começam com 0x, exemplo: 0x12A4, e as constantes de base oito começam com zero (0), exemplo: 01342. Cuidado, nunca digite 021 para referenciar o número 21, o compilador C considera isto como 17 (21 em base 8). Uma constante caracter deve ser inserida como ‘c’.

A forma “c”, com aspa dupla, é utilizada para inserir uma string constante, neste caso, é um vetor de dois caracteres, o ‘c’ e o ‘\0’ (caracter nulo finalizando a cadeia). Como o último caracter constante, existem várias outras constantes de barra invertida que tem significado especial. Alguns deles são:
\b – retrocesso, \n – nova linha, \t – tabulação horizontal, \v – tabulação vertical, \r – retorno de carro, \a – alerta(sinal sonoro), \” – aspas, \’ – apóstrofo, \\ - barra invertida.

Uma **constante declarada** é formada ao anteceder a palavra-chave **const** à definição normal de uma variável. Observar que deste jeito pode-se especificar tipo de dado, terminar com ponto e vírgula e inicializar a constante. Exemplo: const notamaxima = 10;

Uma **constante definida** é dada utilizando uma macro **#define**. Neste caso, a constante é dada pelo pré-processador no código substituindo o nome definido pelo valor de definição. Observe que #define não especifica tipos de dados, não utiliza o símbolo de atribuição (=) e nem termina com ponto e vírgula. Exemplo : #define MAXIMANOTA 10

Ao invés de definir constantes individuais que recebem valores subjacentes, pode-se utilizar **constantes enumeradas** para criar listas categorizadas que atingem o mesmo objetivo. Utilizando a palavra chave **enum** diz-se ao compilador que os itens dados devem ser enumerados, isto é, associados a números seqüenciais iniciando com o zero, quando nada é especificado, mas pode atribuir-se um valor expresso e os seguintes assumem valores consecutivos. Um elemento de uma enumeração não tem endereço de armazenamento em memória. Exemplos,

```
enum CORES {vermelho, amarelo, azul, laranja, verde, violeta} cor;

enum STATUS{FALSE, TRUE, FAIL=0, OK, NOT_RUN=-1};
STATUS estado;
```

Tipo de armazenamento das variáveis

O **escopo** de uma variável refere-se aos limites de validade de uma variável. Apenas no escopo de uma variável as instruções podem-se referir à variável.

Como mencionado acima, antes de utilizar uma variável ela deve ser declarada. Nesta declaração é estabelecido um identificador e um tipo de dado para a variável. Além disso, pode-se especificar a forma de armazenamento da variável, isto tem a ver com seu escopo e se o armazenamento é na memória ou nos registros do CPU.

Uma **variável automática** é armazenada na memória e seu escopo é limitado ao bloco no qual aparece, isto é, enquanto esse bloco ou qualquer um outro bloco mais interior ao atual está sendo executado a variável existe. Quando o bloco onde a variável automática foi declarada acaba a variável deixa de existir. Exemplo : auto int segundos;

Uma **variável registro** é uma variável automática que pode ser armazenada nos registros do CPU, se existir algum registro livre e de tamanho suficiente para armazenar a variável. As variáveis armazenadas nos registros do CPU são bem mais rápidas, embora devam ser utilizadas apenas para as variáveis do seu programa que são muito utilizadas. O número de registros do CPU disponíveis é limitado. Exemplo : register int segundos;

Uma **variável estática** são locais como as variáveis automáticas. A diferença é que as variáveis estáticas não desaparecem quando acaba o bloco ou a função onde ela foi declarada, o valor da variável persiste mesmo que não seja disponível. Se retorna-se ao bloco ou a função onde foi declarada uma variável estática, ela fica novamente disponível e com o último valor armazenado nela. As variáveis estáticas inicializam-se apenas uma vez no tempo de compilação, portanto ocupam memória mesmo que não estejam ativas. Exemplo : static int segundos;

Uma **variável externa** tem escopo global, isto é, existem em qualquer bloco e estão disponíveis a qualquer função que necessite utilizá-la. Ela é acessível a todas as instruções não, importando onde estejam localizadas. Exemplo : extern int segundos;

Em resumo, as variáveis automáticas são locais aos seus próprios blocos ou funções e os seus valores desaparecem quando o bloco ou função termina. As variáveis estáticas são também locais mas os seus valores persistem, e as variáveis externas são globais e os seus valores também persistem.

A linguagem C é pragmática, daí que admite valores por default, valores que são assumidos na falta de especificação. A forma de armazenamento de uma variável tem valor por default: o compilador assumirá uma forma de armazenamento pelo contexto quando nada é especificado. Isto significa que o valor por default não é único. Se uma variável é definida em um bloco ou função e não é especificada a forma de armazenamento é assumida automática, mas se foi declarada fora de qualquer bloco ou função ela será assumida externa pelo compilador.

```
/* Comentário adequado */

#include <stdio.h>

int value;

main( )
{
    int somando = 20;

    value = 350;
    value = value + somando;

    {
        int vezes;

        value = value + vezes * somando;

        {
            double Imposto;

            Imposto = .12 * value;
            printf("Valor do Imposto = %lf
\n",Imposto);

        }
        value = 0;

    }
    printf("Fim do programa.\n");

}
```

Operadores aritméticos e de atribuição

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

Operador	Ação
+	Soma de inteiros e pontos flutuantes
-	Subtração de inteiros e ponto flutuante ou troca de sinal
*	Multiplicação de inteiros e pontos flutuantes
/	Divisão de inteiros e pontos flutuantes
%	Resto de divisão de inteiros
++	Incremento de inteiros e pontos flutuantes
--	Decremento de inteiros e pontos flutuantes

O C possui operadores aritméticos unários e binários. Os unários agem sobre uma variável apenas, modificando ou não o seu valor, e retornam o valor final da variável. O operador – como troca de sinal é um operador unário que retorna o valor da variável multiplicado por -1. Os operadores de incremento e decremento são unários, incrementando e decrementando de 1 a variável sobre a qual está aplicado. Estes operadores podem ser pré-fixados ou pós-fixados. Quando são pré-fixados eles incrementam e retornam o valor da variável já incrementada. Quando são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável. Exemplos :

1. x++; equivalente a x = x+1;
2. Em x = 23;
   y = x++;
   no final tem-se y = 23 e x = 24.
3. Em x = 23;
   y = ++x;
   no final tem-se y = 24 e x = 24.

Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais. A soma, subtração, multiplicação, divisão e resto são operadores binários pois pegam duas variáveis, somam, subtraem, multiplicam, dividem ou acha o resto dos seus valores sem alterar as variáveis, e retornam este resultado.

O operador de atribuição do C é o sinal de igual, =. O que ele faz é pegar o valor à direita e atribuir à variável da esquerda. Além disto ele retorna o valor que atribuiu à esquerda, assim sendo são válidas as seguintes sentenças :

```
double x, valor, custo;
x = valor = custo = 123.5;
```

Ao contrário de outras linguagens, o operador de atribuição em C pode ser utilizado em expressões que envolvem outros operadores, formando os chamados operadores de atribuição compostos. Os operadores compostos condensam sentenças de atribuição da forma :

Variável = *Variável* **operador** *expressão*;

na forma :

Variável **operador =** *expressão*;

Exemplo :

valor = valor + custo;

escreve-se na forma :

valor += custo;

Os operadores atribuição podem ser compostos com os operadores aritméticos (veja a primeira linha da tabela) e podem ser compostos com os operadores bit a bit (segunda linha da tabela).

+=	-=	*=	/=	%=
>>=	<<=	&=	|=	^=

Operadores Relacionais e Lógicos

Os operadores relacionais do C realizam comparações entre variáveis. Relação refere-se as relações que os valores podem ter um com o outro. Os operadores relacionais são :

>	Maior do que
>=	Maior ou igual
<	Menor do que
<=	Menor ou igual
==	Igual a
!=	Diferente de

Os operadores relacionais retornam o valor 1 para verdadeiro e retornam 0 para falso.

Os operadores lógicos fazem operações com valores lógicos, isto é, verdadeiro e falso. Verdadeiro é qualquer valor diferente de zero (0), enquanto zero é falso. As operações de avaliação produzem um resultado zero ou um. Os operadores lógicos são:

&&	o **E** lógico
||	o **OU** lógico
!	Negação (unário não)

Operadores Lógicos Bit a Bit

O C permite que se faça operações lógicas "bit a bit" em números. Uma operação bit a bit refere-se a testar, atribuir, ou deslocar os bits efetivos em um byte. Operações bit a bit não podem ser usadas em float, double, long double, void ou outros tipos mais complexos. Estas operações são aplicadas aos bits individuais dos operandos, ou seja, o número é representado por sua forma binária e as operações são feitas em cada bit dele. Lista de operadores bit a bit.

&	e bit a bit
|	ou bit a bit
^	ou exclusivo bit a bit
~	negação bit a bit
>>	deslocamento de bits a direita
<<	Deslocamento de bits a esquerda

Imagine uma variável inteira de 16 bits, **i = 6**. A representação binária da i é: 0000000000000110. Operando bit a bit com a negação do número, **~i**, o número se transforma em: 1111111111111001.

Exemplos: Sejam as variáveis

unsigned int itens = 5684; em binário é 0001011000110100
unsigned int alunos = 645; em binário é 0000001010000101
Resultado de : itens & alunos é 0000001000000100
itens | alunos é 0001011010110101
itens ^ alunos é 0001010010110001

Tenha muito cuidado no uso dos operadores bit a bit, principalmente para não confundi-los com os operadores lógicos, pois os operadores lógicos sempre retornam o resultado zero ou um, enquanto os operadores bit a bit similares produzem um valor de acordo com a operação especificada. Por exemplo, se na variável inteira é atribuído x = 7; então x && 8 é verdadeiro, enquanto que x & 8 é falso. Porquê?

Os operadores de deslocamento << e >> movem todos os bits de uma variável para a direita ou para a esquerda, como especificado. A forma geral de deslocamento é :

variável >> número de posições de bits.

Conforme os bits são deslocados para uma extremidade, zeros são colocados na outra. Lembre-se de que um deslocamento não é rotação, isto é, os bits que saem por uma extremidade não voltam para a outra. Se uma variável inteira i = 3; é deslocada por i << 3; o valor da variável i atualmente é 24.

Operadores vírgula e sizeof

O operador vírgula **,** é usado para encadear diversas expressões que devem ser executadas em forma seqüencial. O valor de uma expressão com o operador vírgula é dado pela expressão mais a direita, isto é útil quando utilizado com um operador atribuição, veja o exemplo seguinte:

```
itens = ( numero = 3, numero++, 2*numero);
```

primeiro atribui o 3 para numero, depois numero é incrementado para 4 e no final atribui 8 a itens. Os parêntesis são necessários o operador atribuição tem precedência sobre o operador vírgula.

O operador **sizeof** é unário e retorna o tamanho em bytes da variável. Exemplo

```
int  itens;
itens = sizeof(itens);      em itens foi atribuído o valor 2.
```

Este operador é usado para gerar códigos portáveis que dependem do tamanho dos tipos de dados.

Precedência e associação

Precedência refere-se à ordem em que o C avalia os operadores quando existem dois ou mais deles em uma sentença. O C tem um conjunto de regras incorporadas para determinar a ordem em que os operadores são avaliados, e é preciso decorá-las para redigir códigos que realizem corretamente as operações. Dizer que um operador tem precedência maior que um outro operador significa que será avaliado antes. Como exemplo, os operadores de relação e lógicos tem a precedência menor que os operadores aritméticos.

A associação refere-se à ordem de avaliação de operadores de igual precedência. Eles podem ser avaliados primeiro de direita à esquerda ou de esquerda à direita.

A seguir está uma tabela com os operadores, a sua precedência e a associação respectiva. A ordem de precedência é de cima para baixo, sendo avaliados primeiro aqueles que estão mais acima.

Precedência	Associação
( ) [ ] . ->	Da esquerda para a direita
! ~ -(unário) ++ -- *(unário) &(unário) (cast) sizeof	Da direita para a esquerda
* / %	Da esquerda para a direita
+ -	Da esquerda para a direita
<< >>	Da esquerda para a direita
< <= > >=	Da esquerda para a direita
== !=	Da esquerda para a direita
&	Da esquerda para a direita
^	Da esquerda para a direita
|	Da esquerda para a direita
&&	Da esquerda para a direita
||	Da esquerda para a direita
? :	Da direita para a esquerda
= += -= *= /= %= (operadores de atribuição)	Da direita para a esquerda
,	Da esquerda para a direita

O problema da precedência e a associação é superado utilizando parênteses no código, e apenas decorando as precedências mais importantes.

Introdução às funções

Uma função é um bloco de código que funciona como uma unidade autônoma para cumprir uma tarefa particular no código do programa. Elas podem ser utilizadas diversas vezes no código do programa e geralmente consistem de chamadas a outras pequenas funções. As funções são concisas, tornando os conceitos fáceis de entender, permitindo que o programa fique mais legível, mais bem estruturado.

Para realizar determinadas ações uma função pode precisar que sejam fornecidos determinados valores. Estes valores(parâmetros) são passados à função nos chamados **argumentos** da função que são as entradas que a função recebe.

Por outro lado, muitas vezes é necessário fazer com que uma função retorne um valor, talvez o resultado das operações particulares que ela realiza. No C, as funções tem um **tipo de retorno**, que é o tipo do valor retornado, isto é, pode ser *void, char, int, long, double*. Quando o tipo de retorno não é especificado, as funções retornam um inteiro. Para dizer ao compilador C o que vai ser retornado precisa-se da palavra-chave **return**.

Modelagem de função : Embora sejam permitidas outras variações, sempre que possível utilize o modelo seguinte para declarar uma função.

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

O nome_da_função é qualquer nome que tenha significado. Se possível, deve descrever a função. A lista_de_argumentos contém um número finito qualquer de argumentos separados por vírgulas. Cada argumento é dado indicando o tipo do argumento e o nome dele.

Agora, fica fácil fazer uma função para multiplicar dois números. No próximo exemplo, o programa utiliza três funções. A função main( ), a função produto( ) e a função printf( ) que já está definida na biblioteca padrão de entrada e saída.

```
#include <stdio.h>

float produto(float  fator1, float  fator2)
{
    return (fator1*fator2);
}

main ( )
{
    float  saida;
    float  coeficiente = 14.2;

    saida = produto(coeficiente, 67.23);

    printf ("Produto = %f\n", saida);
}
```

Na definição da função produto( ) diz-se que a função receberá dois argumentos float. Quando é chamada a função produto( ), são passados como argumentos a variável coeficiente e o número 67.23. Há alguns pontos a observar na chamada a função. Em primeiro lugar tem-se de satisfazer aos requisitos da função quanto ao tipo e à quantidade de argumentos. Em segundo lugar, não é importante o nome da variável que se passa como argumento, pois, a variável coeficiente ao ser passada como argumento para produto( ) é copiada para a variável fator1. Dentro de produto( ) trabalha-se apenas com fator1, então, se fosse mudado o valor do fator1 dentro de produto( ) o valor de coeficiente na função principal main( ) permanece inalterado. Faça o teste.

Repare que os argumentos são separados por vírgula e que deve-se explicitar o tipo de cada um dos argumentos, um a um. Observe 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, neste caso o número 67.23 é copiado para fator2. A função main( ) deve retornar um inteiro pois o tipo de retorno dela não foi especificado, mas como não é feito nenhum retorno, no momento da compilação deve aparecer uma mensagem "warning" (cuidado). Acrescente a linha *return 0;* e compile novamente. Observe a diferença entre o valor de retorno da função main( ) e a saída procurada pelo programa, o produto de 14.2 e 67.23.

Antes de estudar mais sobre as funções no C, apresentam-se várias funções básicas da biblioteca de entrada e saída padrão do C.

Biblioteca padrão de entrada e saída

Uma biblioteca contém o código objeto de uma coleção de funções. As bibliotecas são semelhantes a arquivos objetos comuns, com a diferença que apenas parte do código na biblioteca é acrescentado a um programa, apenas o código das funções da biblioteca chamadas no programa.

A biblioteca padrão C é fornecida com o compilador. Esta biblioteca padrão define um conjunto grande e diversificado de funções. A variedade e a flexibilidade dela põe o C à frente de muitas linguagens de programação. Várias funções na biblioteca padrão trabalham com seus próprios tipos de dados específicos, e estes tipos de dados são definidos nos chamados arquivos de cabeçalho fornecidos e os quais devem ser incluídos no arquivo que utilize essas funções.

Os arquivos de cabeçalho mais utilizados são:

- ctype.h para manipulação de caracteres
- math.h com grande variedade de funções matemáticas
- stdio.h para entrada e saída padrão e de/para arquivos
- stdlib.h diversas declarações
- string.h suporta funções de cadeias de caracteres(strings).

Restringe-se por enquanto o estudo das principais funções da biblioteca de entrada e saída, as que podem ser utilizadas incluindo o arquivo de cabeçalho *stdio.h*.

printf(...)

É uma função que permite escrever no dispositivo padrão de saída, geralmente mostra na tela. A sintaxe dela é a seguinte:

printf("*expressão_de_controle*", lista_de_argumentos);

A expressão_de_controle, não é necessariamente o que será mostrado na tela, mas também descreve tudo que a função vai colocar na tela, isto é, quais as variáveis e suas respectivas posições. Para indicar as posições dos valores das variáveis utiliza-se a notação %, e junto um código de formatação indicando o formato em que a variável deve ser impresso nessa posição. O nome da variável a ser apresentada deve ser dada na lista de argumentos. É muito importante que, para cada caractere de controle %, exista um argumento na lista de argumentos. Os argumentos são separados por vírgulas.

Caractere de formatação	Significado
%c	Caractere simples
%d	Inteiro
%e	Notação científica
%f	Ponto flutuante (float)
%lf	Ponto flutuante(double)
%s	Cadeia de caracteres

Exemplo:

```
#include <stdio.h>
main( )  {
    printf("Este é o numero dois: %d",2);
    printf("\n\t%s está a %d milhões de milhas do sol", "Vênus", 67);

    printf ("\nTeste %% %%\t\t");
    printf ("%f\n",40.345);
    printf ("Um caractere %c e um inteiro %d\n",'D',120);
    printf ("%s e um exemplo\n","\nEste");
    printf ("%s%d%%\n","Juros de ",10);
}
```

Os próximos exemplos mostram, na ordem, como arredondar, alinhar, indicar tamanho dos campos na impressão e imprimir caracteres.

```
#include <stdio.h>
main( )  {
    printf("\n%4.2f",3456.781);
    printf("\n%3.2f",3456.781);
    printf("\n%3.1f",3456.78);
    printf("\n%10.3f",3456.78);
}
```

```
#include <stdio.h>
main( )  {
    printf("\n%10.2f %10.2f %10.2f",8.0,15.3,584.13);
    printf("\n%10.2f %10.2f %10.2f",834.0,1500.55,4890.21);
}
```

```
#include <stdio.h>
main( )  {
    printf("\n%2d",350);
    printf("\n%4d",350);
    printf("\n%6d",350);
    printf("\n%04d",21);
    printf("\n%06d",21);
    printf("\n%6.4d",21);
    printf("\n%6.0d",21);
}
```

```
#include <stdio.h>
main( )  {
    printf("%d %c %x %o\n",'A','A','A','A');
    printf("%c %c %c %c\n",'A',65,0x41,0101);
}
```

A formato ASCII possui 256 códigos de 0 a 255. Se é impresso em formato caractere um número maior do que 255, será impresso o resto da divisão do número por 256; se o número for 3393 será impresso ‘A’ pois o resto de 3393 por 256 é 65.

scanf(...)

Ela é o complemento de printf(...) e nos permite ler dados formatados da entrada padrão (teclado), isto é, com ela pode-se pedir dados ao usuário. A sintaxe é

scanf (“*expressão_de_controle”*, lista_de_argumentos);

O número de variáveis na lista de argumentos deve ser o mesmo que o número de códigos de formatação na expressão_de_controle. Outra coisa importante, a lista de argumentos deve consistir dos endereços das variáveis para armazenamento dos dados lidos. Isto é feito colocando o símbolo **&** antes de cada nome de variável na lista de argumentos.

O símbolo & faz referência ao **operador de endereço** que retorna o endereço na memória da variável de tipo básico do C. Para entender isto, deve-se saber que a memória do computador é dividida em bytes, e eles são numerados de 0 até o limite da memória. O número que corresponde ao primeiro byte ocupado pela variável é chamado de endereço da variável.

```
#include <stdio.h>
main( ) {
    int numero1;
    double numero2;
    printf("Digite um número = ");
    scanf("%d",&numero1);
    printf("\nO número é %d",numero1);
    printf("\no endereço e %u",&numero1);
    printf(“\nDigite um segundo número = “);
    scanf(“%lf”,&numero2);
    printf(“O segundo número é %lf, ou %f, ou %d\n.”,numero2,numero2,numero2,);
}
```

getchar( )

É a função original de entrada para um caractere dos sistemas baseados em UNIX. O caractere é armazenado pelo getchar( ) até que a tecla de retorno (enter) seja pressionada.

```
#include <stdio.h>
main( ) {
    char c;
    c = getchar( );
    printf(“O caractere inserido é %c\n”,c);
}
```

Existem muitas variantes da getchar( ) bem mais úteis em determinadas situações, como a getch( ) ou a getche( ) que são encontradas no arquivo de cabeçalho conio.h, mas este arquivo foi implementado para ambiente DOS ou Windows não para ambiente UNIX, e não pertence ao C padrão.

putchar(int c**)**

Escreve na tela o argumento de seu caractere na posição corrente. O argumento pode ser um inteiro que é transformado em caractere ou pode ser um simples caractere.

```
#include <stdio.h>
main( ) {
    char c = 'D';
    printf("\nA variável c foi inicializada com o valor %c, isto é %d.",c,c);
    printf("\nDigite uma letra minúscula  ");
    c = getchar();
    putchar(toupper(ch));
    putchar('\n');
}
```

Há inúmeras outras funções de manipulação de char complementares às que foram vistas, como isalpha( ), isupper( ), islower( ), isdigit( ), isspace( ), toupper( ), tolower( ), que são encontradas no arquivo de cabeçalho ctype.h.

Visando utilizar eficientemente estas funções de entrada e saída de dados explicita-se melhor alguns aspectos relativos aos caracteres e às cadeias de caracteres ou strings. Os caracteres ou variáveis de tipo char são tratados pelo C como variáveis de um byte (8 bits). Os inteiros ou variáveis do tipo int têm um número maior de bytes, dependendo da implementação do compilador eles têm 2 ou 4 bytes. Assim sendo, um char pode armazenar tanto valores numéricos inteiros de 0 a 255 quanto um caractere de texto. Para indicar um caractere de texto usamos apóstrofes. Veja o exemplo anterior.

Como visto em vários exemplos anteriores, muitas vezes é necessário manipular vários caracteres juntos e consecutivos como uma seqüência de caracteres já conhecida como string. Para declarar uma string utiliza-se o seguinte formato geral

char nome_da_string[tamanho_da_string];

Isto significa declarar um vetor com tamanho_da_string variáveis do tipo char. Uma particularidade de uma string é que ela termina com o caractere nulo '\0', cujo valor é igual a zero. Deve-se declarar o comprimento de uma string como sendo, no mínimo, um caractere maior que a maior string que se pretende armazenar.

Exemplo. Para declarar uma string de 7 posições escreve-se

```
char  nome_amigo[7];
```

inserindo o nome Luis na string, o vetor nome_amigo tem a forma :

L	u	i	s	\0	?	?

As duas últimas células do vetor que não foram utilizadas têm valores indeterminados, porque o C não inicializa variáveis, cabendo ao programador esta tarefa. Para ler uma string fornecida pelo usuário pode-se usar a função gets( ), que coloca o caractere '\0' no final quando o usuário aperta a tecla "Enter".

```
#include <stdio.h>
main ( )
{
    char string[100];
    printf ("Digite uma string: ");
    gets (string);
    printf ("\n\nVoce digitou %s",string);
    printf ("\n\nO terceiro caractere digitado é %c. Modifique-o.",string[2]);
    printf ("\n\tIngresse um caractere ");
    scanf ("%c", &string[2]);
    printf ("\n\nO terceiro caractere atualmente é %c.",string[2]);
}
```

Para se acessar um determinado caracter de uma string, basta utilizar o índice ou número de posição do caractere na string. Lembre-se de que os índices nos vetores no C sempre começam em zero.

Pode-se inicializar a string no momento da sua declaração utilizando uma **string constante**, que é dada sempre entre aspas. Por exemplo,

```
char  nome_amigo[7] = "Luis";
```

Estruturas de controle

Os comandos de controle de fluxo são a essência de qualquer linguagem, porque governam o fluxo da execução do programa. São poderosos e ajudam a explicar a popularidade da linguagem. As estruturas de controle de fluxo são fundamentais porque sem elas só haveria uma maneira do programa ser executado, de cima para baixo e comando por comando. Não seria possível testar condições, fazer repetições ou saltos. A linguagem C possui diversos comandos de controle de fluxo. É possível resolver todos os problemas sem utilizar todas elas, mas deve-se manter a elegância e facilidade de entendimento fazendo bom uso das estruturas no local certo.

Podemos dividir em três categorias. A primeira consiste em instruções condicionais if, else-if, switch e o condicional ?:. A segunda são os comandos de controle de loop: o while, for e o do-while. A terceira contém instruções de desvio incondicional goto (a menos elegante).

Instruções condicionais

O Comando if

É usado para executar condicionalmente um segmento de código, isto é, apenas quando é satisfeita uma condição é executada uma parte do código. Existe também o comando else, que poder ser pensado como complemento do comando if. Quando a condição em if não é satisfeita é ativado o else executando uma outra parte do código.

Sintaxe:

```
if (condição) {
    bloco_de_comandos_if;
}
else {
    bloco_de_comandos_else;
}
```

Observe que existem dois blocos de comandos, um bloco para o if e um outro bloco para o else.

A *condição* deve ser VERDADEIRA(não zero) ou FALSA(zero). Se a condição é avaliada como verdadeira, o computador executará o comando ou o bloco_de_comandos do if. Se a condição é falsa, caso a cláusula else existir, o computador executará o comando ou o bloco_de_comandos do else ignorando os comandos do if, que não serão executados.

```
#include <stdio.h>
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");
}
```

```
#include <stdio.h>
main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num==10) {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.\n");
    }
   else {
       printf ("\n\nVoce errou!\n");
       printf ("O numero e diferente de 10.\n");
    }
}
```

É importante nunca esquecer que, quando usamos a estrutura if-else, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas.

O comando if-else-if

A estrutura if-else-if é apenas uma extensão da estrutura if-else, para testar diferentes condições. Sua forma geral pode ser escrita como sendo:
Sintaxe:

```
if (condição_1) {
    bloco_de_comandos_1;
}
else if (condição_2 ) {
```

```
    bloco_de_comandos_2;
}
else if (condição_3) {
    bloco_de_comandos_3;
}
.
.
else {
    bloco_de_comandos_default;
}
```

A estrutura acima funciona da seguinte maneira, o programa começa testando a condição_1. Se ela for verdadeira, executa o bloco_de_comandos_1 e cai fora de todos os outros elses. No caso contrário, testa a condição_2 no primeiro else if; se ela for verdadeira, executa o bloco_de_comandos_2 e cai fora dos próximos elses. Se a condição_2 também for falsa, testa a condição_3 do próximo else if, se ela for verdadeira, executa o bloco_de_comandos_3 e cai fora dos próximos elses. Caso contrário, testa a condição do próximo else if e assim sucessivamente. Apenas quando todas as condições são falsas é executado o bloco_de_comandos_default. Só um bloco_de_comandos será executado, isto é, só será executada a declaração equivalente à primeira condição que resultar diferente de zero.

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

```
#include <stdio.h>
main ( )
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num==10) {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.\n");
```

```
    }
    else {
        if (num>10) {
            printf ("O numero e maior que 10.");
        }
        else {
            printf ("O numero e menor que 10.");
        }
    }
}
```

É possível ter um if dentro da declaração de um outro if mais externo, como no último exemplo. Isto é chamado de ifs aninhados. Porém, você deve saber exatamente a qual if um determinado else está ligado.

Switch

Pode ocorrer que você queira testar uma variável ou uma expressão em relação a vários valores. Como visto acima, pode-se utilizar if-else-if, mas também existe uma outra opção, o comando switch.

Uma instrução switch torna-se prática sempre que um programa necessita selecionar algumas ações dentre as diversas possíveis, tendo como base o resultado de uma expressão ou uma variável, equivalentes a um valor inteiro ou a um caractere. Isto é, o comando switch é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos.

Sintaxe:

```
switch (variável) {
    case constante_1:
        bloco_de_comandos_1;
        break;
    case constante_2:
        bloco_de_comandos_2;
        break;
    .
    .
    case constante_n:
        bloco_de_comandos_n;
        break;
    default:
        bloco_de_comandos_default;
}
```

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

O comando break faz com que o switch seja interrompido assim que uma das declarações for executada. Mas ele não é essencial ao comando switch. Se após a execução do bloco_de_comandos não houver um break, o programa continuará executando os próximos blocos de comandos sem testar com o valor das próximas constantes. Isto pode ser útil em algumas situações, mas é recomendado muito cuidado.

```
#include <stdio.h>

main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);

    switch (num) {
        case 9:
            printf ("\n\nO numero e igual a 9.\n");
            break;
        case 10:
            printf ("\n\nO numero e igual a 10.\n");
            break;
        case 11:
            printf ("\n\nO numero e igual a 11.\n");
            break;
        default:
            printf ("\n\nO numero nao e nem 9 nem 10 nem 11.\n");
    }

}
```

O condicional ? :

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

```
int num;

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

equivale a

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

Com isto, é possível simplificar algumas expressões simples.

A instrução condicional **? :** proporciona uma maneira rápida de se escrever uma condição de teste. Como nas anteriores, instruções condicionais são verificadas e ações associadas são executadas conforme a expressão for avaliada como verdadeira ou falsa.

Sintaxe:

(*condição*) **?** *ação_1* **:** *ação_2*;

Quando avaliada a condição como verdadeira, é executada a ação_1, caso contrário é executada a ação_2. Isto significa que a instrução condicional anterior é equivalente ao if-else seguinte:

```
if (condição) {
    ação_1;
}
else {
    ação_2;
}
```

O operador ? : também é conhecido como operador ternário porque precisa de três operandos. O operador ? : é limitado pois não atende a uma gama muito grande de casos.

```
#include <stdio.h>
main( ) {
    char c;
    int resp, val1, val2;
    printf("\nDigite dois valores inteiros : ");
    scanf("%d %d", &val1, &val2);
    printf("\nDigite '+' para somar e outra tecla para substrair : ");
    scanf("%c", c);
    resp = (c == '+') ? val1+val2 : val1-val2;
    return 0;
}
```

Instruções em loops

A linguagem C contém uma série-padrão de instruções de controle de repetição, os chamados laços **for, while, do-while**, que compõem a segunda família de comandos de controle de fluxo. Todos os laços podem terminar naturalmente baseados na condição de teste booleano. No entanto em C, um laço de repetição pode terminar devido a uma condição de erro antecipado usando instruções como break, ou exit. Os laços de repetição podem também ter seu fluxo de controle lógico alterado por instruções break e continue.

O laço for

O comando for é utilizado para repetir um comando ou um bloco de comandos diversas vezes, de maneira que se possa ter um bom controle sobre o loop ou laço.
Sintaxe:

```
for ( inicialização; condição; incremento) {
    bloco_de_comandos;
}
```

Em sua forma mais simples, a inicialização é um comando de atribuição que o compilador usa para estabelecer a variável de controle do laço. A condição é uma expressão de relação que testa a variável de controle do laço contra algum valor para determinar quando terminará as repetições. O incremento define a maneira como a variável de controle do laço será alterada cada vez que o computador repetir o laço.

Para entender melhor o laço **for,** observe como ele funciona. O laço for é equivalente a se fazer o seguinte:

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

Assim, 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 o bloco_de_comandos, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Um ponto importante é que podemos omitir qualquer uma das três expressões do for, isto é, se não necessita-se de uma inicialização pode ser omitida.

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

O incremento da variável count é feito usando o operador de incremento. Esta é a forma usual de se fazer o incremento (ou decremento) em um laço for.

Como nenhuma das três expressões no laço é necessária, pode-se ter a forma : **for(; ;)**. Neste caso não existe inicialização, nem teste de condição, nem incremento.

Quando não existe teste de condição expressa, o for assume condição sempre verdadeira, o que implica executar o laço para sempre, a não ser que ele seja interrompido. Esta forma é chamada de **loop infinito**. Para interromper um loop como este usa-se o comando break. O comando break vai quebrar o loop infinito e o programa continuará sua execução normalmente. Como exemplo veja-se o programa que faz a leitura de uma tecla e sua impressão na tela, até que o usuário aperte uma tecla especial, denominada FLAG. O nosso FLAG será a letra 'X'.

```
#include <stdio.h>
main ()
{
    int Count;
    char ch;
    for ( Count=1; ; Count++ ) {
        fflush(NULL);
        scanf("%c",&ch);
        if (ch  ==  'X')  break;
        printf("\nLetra: %c\n",ch);
    }
}
```

Atenção ao comando fflush(NULL). O papel deste comando é limpar o buffer do teclado para que outros caracteres armazenados no buffer do computador sejam liberados. Desta forma a leitura de caractere que acontece logo após a sua execução não ficará prejudicada.

Caso o bloco_de_comandos seja vazio, diz-se que o laço é sem conteúdo. Um loop sem conteúdo tem a forma (observe o ponto e vírgula!) : **for (** *inicialização; condição; incremento***) ;**
Uma das aplicações desta estrutura é gerar tempos de espera.

```
#include <stdio.h>

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

Por outra parte, as expressões no laço for podem ser compostas por várias instruções. Observe.

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

Combinando o operador ? : com o for, mostra-se um exemplo interessante, do contador circular.

```
#include <stdio.h>
main( )
{
    int index = 0, counter;
    char message[5] = "Curso C";
    for (counter = 0; counter < 1000; counter++) {
        printf("%c", message[index]);
        index = (index  ==  7) ? (index=0; printf("\n");) :  ++index;
    }
}
```

A mensagem *Curso C* é escrita na tela até a variável counter determinar o término do programa. Enquanto isto a variável index assume os valores 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, ... progressivamente.

O laço while

Uma segunda maneira de executar um laço é utilizando o comando while, que significa enquanto. O while permite que o código fique sendo executado numa mesma parte do programa de acordo com uma determinada condição.

Sintaxe:

```
while (condição) {
    bloco_de_comandos;
}
```

Para o while também o bloco_de_comandos pode ser vazio, simples ou múltiplas instruções. Ele testa a condição antes de executar o laço e executa o bloco_de_comandos desde que a condição seja verdadeira, fazendo o teste novamente e assim por diante. Como o comando for, pode ser utilizado para um loop infinito, bastando para isto dar uma condição eternamente verdadeira como : while(1).

Observe como funciona o while.

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

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

O laço do-while

Difere tanto do loop for quanto do while pelo fato de ser um laço do tipo pós-teste.
Sintaxe:

do {
    bloco_de_comandos;
} while(*condição***);**

A grande novidade no comando do-while é que ele, ao contrário do for e do while, garante que o bloco_de_comandos será executado pelo menos uma vez, pois o primeiro teste é realizado ao final da primeira iteração.

Mesmo que o bloco_de_comandos seja apenas um comando é uma boa prática deixar as chaves. O ponto e vírgula final é obrigatório. O funcionamento do comando do-while é como segue.

Bloco_de_comandos;
if (condição) “volta para o bloco_de_comandos”

Observa-se que a estrutura do-while executa o bloco_de_comandos, testa a condição e, se esta for verdadeira, volta para o bloco_de_comandos. Um dos principais usos da estrutura do-while é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido.

```
#include <stdio.h>
main ( )
{
    char c;
    do {
        printf ("\n\nEscolha uma alternativa :\n\n");
        printf ("\t(1)...Mamao\n\t(2)...Abacaxi\n\t(3)...Laranja\n\n");
        fflush(NULL);
        scanf("%c", &c);
    } while ((c!='1') && (c!='2') && (c!='3'));
    switch (c) {
        case 1:
            printf ("\t\tVoce escolheu Mamao.\n");
            break;
        case 2:
            printf ("\t\tVoce escolheu Abacaxi.\n");
            break;
        case 3:
            printf ("\t\tVoce escolheu Laranja.\n");
            break;
    }
}
```

break, continue

O Comando **break** tem dois usos. É utilizado para quebrar a execução dos comandos switch, e ele interrompe a execução de qualquer loop ou laço. O break é utilizado para sair de um laço antes que a condição de teste se torne falsa e faz com que a execução do programa continue na primeira linha seguinte ao laço.

Algumas vezes torna-se necessário "pular" uma parte do programa. Para isso é utilizado o continue. O **continue** força a próxima iteração do laço e pula o código que estiver em seguida. Ele é diferente do break, apenas funciona dentro de um laço. Quando o comando continue é encontrado, o laço pula para a próxima iteração, mas não sai do laço.

```
#include <stdio.h>
main( )
{
    int opcao;
    while (opcao != 5) {
        printf("\n\n Escolha uma opcao entre 1 e 5: ");
        scanf("%d", &opcao);
        if ((opcao > 5)||(opcao <1)) continue;  /* Opcao invalida: volta ao inicio do loop */
        switch (opcao) {
            case 1:
```

```
                printf("\n --> Primeira opcao..");
                break;
            case 2:
                printf("\n --> Segunda opcao..");
                break;
            case 3:
                printf("\n --> Terceira opcao..");
                break;
            case 4:
                printf("\n --> Quarta opcao..");
                break;
            case 5:
                printf("\n --> Abandonando..");
                break;
        }
    }
}
```

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

Instrução de desvio incondicional

O comando goto

O goto é o último comando de controle de fluxo. Ele pertence a classe dos comandos de salto incondicional.

O goto realiza um salto para um local especificado, que é determinado por um rótulo. Um rótulo, na linguagem C, é uma marca no programa. Você dá o nome que quiser a esta marca. Sintaxe:

nome_do_rótulo :
.
.
.
goto *nome_do_rótulo*;

Deve-se declarar o nome do rótulo na posição a qual se gostaria dar o salto e seguido **:** dois pontos. O goto pode saltar para um rótulo que esteja mais à frente ou para trás no programa, embora o rótulo e o goto devam estar dentro da mesma função.

Observar como funciona o goto quando utilizado para substituir um comando for.

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

Recomenda-se utilizar o comando goto com parcimônia, pois o abuso no seu uso pode tornar o código confuso. O goto não é um comando necessário, mas o seu bom emprego pode facilitar o entendimento de algumas funções. O comando goto pode tornar um código muito mais fácil de se entender se ele for bem empregado. Um caso em que ele pode ser útil é quando temos vários loops e ifs aninhados e se queira, por algum motivo, sair destes loops e ifs todos de uma vez. Neste caso um goto resolve o problema muito mais elegantemente que vários breaks, sem contar que os breaks exigiriam muito mais testes. Neste caso o goto é mais elegante e mais rápido. Não abusar.

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

Vetores, Matrizes e Strings

Vetores

Os computadores processam dados. Geralmente os dados são organizados em forma ordenada, por exemplo em uma série temporal, na saída de um experimento que varia quando é modificada a entrada, ou em uma lista de nomes em ordem alfabética. Uma coleção de dados tais que os seus elementos formam uma seqüência ordenada é chamada de array ou vetor de dados. Os elementos deste vetor podem ser de qualquer tipo de dado, isto é, pode-se ter um vetor de inteiros, de caracteres, etc. A seguir estuda-se as características, notação, manuseio, diferentes tipos dos vetores.

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

tipo_da_variável **nome_da_variável_vetor** [tamanho];

Na presença de uma declaração como esta o C reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho, e todas elas com o espaço suficiente para armazenar valores do tipo tipo_da_variável.

tamanho – deve ser um valor constante.

Ao declarar : **float exemplo [20]**; o C irá reservar 20 espaços para pontos flutuantes do tipo double, isto é 4x20=80 bytes. Estes bytes são reservados de maneira contígua. Na linguagem C a numeração começa sempre em zero. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessar os elementos do vetor, é suficiente chamar o nome_da_variável_vetor e entre colchetes indicar o número do elemento desejado menos um. Em **exemplo[0]** e **exemplo[9]**, está-se acessando o primeiro elemento e o décimo elemento do vetor **exemplo.**

O C não verifica o índice que o programador informa para acessar os elementos de um vetor. Se o programador não tiver cuidado com os limites de validade para os índices ele corre o risco de ter sobre-escritas ou de estar alocando fora do vetor e talvez ver o computador travar. Assim, ninguém impede que se escreva, exemplo[30];.

```
#include <stdio.h>

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

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

Strings

Strings são vetores de chars, ou caracteres. As strings são o uso mais comum para os vetores. Fique sempre atento para o fato de que as strings têm o seu último elemento como sendo um '\0'. A declaração geral para uma string é:

char **nome_da_string** [tamanho];

As strings foram já estudadas anteriormente, mas o seu estudo é ampliado aqui, pois a biblioteca do C possui diversas funções que manipulam strings. Estas funções são úteis pois não é possível, por exemplo, igualar duas strings : string1 = string2; /* Nunca faça isto */ No capítulo dos ponteiros se verá porque. Por enquanto aprenda que as strings devem ser igualadas elemento a elemento, isto é, faz-se a cópia dos caracteres de uma string para o vetor da outra string. O caractere '\0' que finaliza toda string pode ser aproveitado em muitas situações, como no exemplo a seguir.

```
#include <stdio.h>
main ( )
{
    int  count;
    char str1[100],str2[100];
    printf("\nEntre com uma string (finaliza com :)  ");
    for(count = 0; count < 99; count++) {
        char c;
        scanf("%c", &c);
        if( c == ':' )  break;
        str1[count] = c;
    }
    str1[count] = '\0';
    for (count=0;str1[count];count++)
        str2[count]=str1[count];
    str2[count]='\0';
   ....          /* Aqui o programa continua */
}
```

O segundo laço for acima é baseado no fato de que a string que está sendo copiada termina em '\0'. Este tipo de raciocínio é a base do C e você deve fazer um esforço para entender como o programa acima funciona. Quando o elemento encontrado em str1[count] é o '\0', o valor retornado para o teste condicional é falso (nulo). Desta forma a expressão que vinha sendo verdadeira (não zero) torna-se falsa.

Funções básicas para manipulação de strings

gets(...)

A função gets( ) lê uma string a partir do teclado. Sintaxe: **gets** (*nome_da_string*);

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

Observe que é válido passar para a função printf( ) o nome da string. Por outro lado, como o primeiro argumento da função printf() é uma string também é válido escrever : printf (string);

strcpy (...)

A função strcpy() copia a string-origem para a string- destino.
Sintaxe: **strcpy** (*string_destino,string_origem*);

```
#include <stdio.h>
#include <string.h>
main ( ) {
    char str1[100],str2[100],str3[100];
    printf ("Entre com uma string: ");
    gets (str1);
    strcpy (str2,str1);
    strcpy (str3,"Voce digitou a string ");
    printf ("\n\n%s%s",str3,str2);
}
```

strcat (...)

Com a função strcat(...) a string de origem permanecerá inalterada e será anexada ao fim da string de destino. Sintaxe: **strcat** (*string_destino,string_origem*);

```
#include <stdio.h>
#include <string.h>
main ( )  {
    char str1[100],str2[100];
    printf ("Entre com uma string: ");
    gets (str1);
    strcpy (str2,"Voce digitou a string ");
    strcat (str2,str1);
    printf ("\n\n%s",str2);
}
```

strlen (...)

A função strlen() retorna o comprimento da string fornecida. O terminador nulo não é contado. Sintaxe: **strlen** (*string*);

```
#include <stdio.h>
#include <string.h>
main ( ) {
    int size;
    char str[100];
    printf ("Entre com uma string: ");
    gets (str);
    size=strlen (str);
    printf ("\n\nA string que voce digitou tem tamanho %d",size);
}
```

strcmp (...)

A função strcmp() compara a string1 com a string2. Se as duas forem idênticas a função retorna zero. Se elas forem diferentes a função retorna não-zero. Sintaxe: **strcmp** (*string1*,*string2*);

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

Matrizes

Os vetores são matrizes uni-dimensionais. Agora trata-se de matrizes multi-dimensionais.

Matrizes bi-dimensionais

No caso de uma matriz bidimensional, a sua declaração é:

tipo_da_variável **nome_da_variável_matriz** [tamanho1] [tamanho2];

Nesta estrutura, o primeiro índice tamanho1, indexa as linhas da matriz e o segundo índice indexa as colunas da matriz. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia mais rapidamente que o índice à esquerda. Não esqueça que os índices no C variam de zero ao valor declarado, menos um. Também não esqueça que o C não verifica os limites válidos da matriz para o usuário. O programador é responsável por manter os índices na faixa permitida.

```
#include <stdio.h>
main ( )
{
    int matrix [20][10];
    int i,j,count;
    count = 1;
    for (i=0;i<20;i++) {
        for (j=0;j<10;j++) {
            matrix[i][j]=count;
            count++;
            printf( “%d\t”, matrix[i][j]);
        }
        printf(“\n”);
    }
}
```

A matriz é preenchida linha por linha, (0 a 19). Cada linha recebe 10 valores (0 a9), iniciando em 1 e terminando em 200.

Matrizes de strings

Matrizes de strings são matrizes bidimensionais. Imagine uma string. Ela é um vetor. Se é declarado um vetor de strings está-se fazendo uma lista de vetores. Esta estrutura é uma matriz bidimensional de chars. Sintaxe:

c*har* **nome_da_variável** [numero_de_strings][tamanho_das_strings];

Para acessar uma string individual, é só utilizar o primeiro índice. Então, para acessar a segunda string faça, **nome_da_variável** [1].

```
#include <stdio.h>

main ( )
{
    char strings [5][100];
    int count;
    for (count=0;count<5;count++) {
        printf ("\n\nDigite uma string: ");
        gets (strings[count]);
    }
    printf ("\n\n\nAs strings que voce digitou foram:\n\n");
    for (count=0;count<5;count++)
        printf ("%s\n",strings[count]);
}
```

Matrizes multidimensionais

Estendendo o mencionado acima, pode-se declarar matrizes multi-dimensionais de forma simples.
Sintaxe:

tipo_da_variável **nome_da_variável_matriz** [tamanho1] [tamanho2] ... [tamanhoN];

Tem-se declarado uma matriz N-dimensional, e o funcionamento dela é basicamente como os tipos anteriores de matrizes. O índice que varia sempre mais rapidamente é o índice mais à direita.

Inicialização de matrizes

Assim como as variáveis no C podem ser inicializadas, as matrizes também podem ser inicializadas. É suficiente listar os valores dos elementos da matriz entre chaves. Observe:

tipo_da_variável **nome_da_matriz** [tam1][tam2] ... [tamN] = **{ lista_de_valores }**;

A lista de valores é composta por valores (do mesmo tipo_da_variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz.

```
float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matrix [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

O primeiro exemplo inicializa um vetor. O segundo inicializa uma matriz com 1,2,3 e 4 na sua primeira linha; 5,6,7 e 8 na sua Segunda linha e 9,10,11 e 12 na terceira linha.

A seguir são apresentadas formas de inicializar strings. O primeiro e segundo caso são duas formas de inicializar a mesma string, a segunda forma é a forma compacta. O terceiro caso inicializa um vetor de strings.

```
char str [10] = { 'J', 'o', 'r', 'g', ‘e’, '\0' };
char str [10] = "Jorge";
char str_vect [3][10] = { "Jorge", "Ze Maria", "Curso C" };
```

Ao inicializar uma matriz é possível não especificar o tamanho dela a priori. O compilador C, na hora da compilação, verifica o tamanho do que você declarou na inicialização e este mesmo será o tamanho da matriz. Este tamanho não pode ser mais mudado durante o programa. Isto é útil para inicializar uma string sem ficar contando o número de caracteres que é necessário.

```
char message [] = "Curso C - O primeiro contato com o C.";
int matrix [][2] = { 1,2,2,4,3,6,4,8,5,10 };
```

O tamanho de message é 38, enquanto a matriz bi-dimensional é da forma matrix[5][2].

Os ponteiros

A dificuldade que têm os leigos para aprender a mexer com os ponteiros, tem a ver com a terminologia utilizada no C. Por exemplo, quando um programador em C diz que uma certa variável é um ponteiro, o que significa? Como uma variável pode apontar para algo? Quando um programador diz que um ponteiro proporciona um valor, como pode um ponteiro proporcionar um valor? Entretanto, não é difícil de entender os ponteiros pensando neles da seguinte forma.

Uma variável do tipo int guarda valores inteiros. Uma variável do tipo float guarda números de ponto flutuante em precisão simples. Se a variável é do tipo char, ela guarda caracteres. Ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel, ele está armazenando um endereço. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C. Um ponteiro também tem tipo. Quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma empresa. Mesmo que o endereço dos dois locais tenham o mesmo formato : rua, número, bairro, cidade, etc, eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros mas de tipos diferentes. Assim, ponteiros que apontam para inteiros são de tipo diferente daqueles que apontam para variáveis ponto flutuante.

Em resumo, um **ponteiro** é uma variável que contém o endereço de uma outra variável. Para declarar um ponteiro utiliza-se a seguinte sintaxe:

tipo_da_variável_apontada ***nome_da_variável_ponteiro**;

A presença do ***** indica ao compilador que aquela variável chamada nome_da_variável_ponteiro, não vai guardar um valor mas sim um endereço para uma outra variável do tipo especificado. O * não é parte do nome do ponteiro.

Para declarar um ponteiro para uma variável de inteiros utilize : **int *nomept**;
Para declarar dois ponteiros em uma mesma declaração, escreva : **char *ptr1,*ptr2**;

Os ponteiros acima declarados ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Utilizar o ponteiro nestas circunstâncias pode levar a problemas inesperados, geralmente perigosos. Antes de usar um ponteiro, ele deve ser inicializado (apontado para algum lugar conhecido).

Operadores de ponteiros : & e *

Para atribuir um valor a um ponteiro basta igualá-lo a um valor de memória. Mas, todos os endereços das variáveis de um programa são determinados pelo compilador na hora da compilação e alocados novamente na execução. Para recuperar estes endereços existe o operador **&** chamado **operador de endereço**. Assim, um ponteiro pode ser inicializado utilizando o operador de endereço. Exemplo:

```
int count=10;
int *ptr;
ptr = &count;       /*a variável ptr aponta para a variável count*/
```

Observe que os conteúdos são totalmente diferentes, e o fato de atribuir o endereço de count para ptr não altera o valor de count. Agora ptr já pode ser utilizado com segurança.

Desde que os pointers são variáveis, eles podem ser manipulados como qualquer variável. Se ptr1 é um ponteiro e ptr2 é um outro ponteiro, ambos para inteiros, pode-se fazer a seguinte declaração:

```
ptr1 = ptr2;
```

Neste caso se faz o ponteiro ptr1 apontar para o mesmo lugar que ptr2.

Com eles pode-se alterar o conteúdo da variável apontada pelo ponteiro. Para isto, é preciso utilizar o operador *inverso* do operador &, ele é o **operador para desreferencia *.** Seu operando é o endereço de uma variável e ele retorna o valor armazenado na variável, é um operador unário. O símbolo * é o mesmo que do operador multiplicação, mas são diferentes operadores. Exemplo:

```
int  itens = 15 + *ptr;
*ptr = 45;
```

Na variável itens é atribuído o valor da variável count acrescentado em 15. Observe que não foi colocado parêntesis, pois os operadores & e * têm maior precedência que os operadores aritméticos. Na segunda instrução foi alterado o conteúdo de count via ponteiro, em count é atribuído o valor 45. Observe que, se ptr aponta para a variável count, então *ptr pode ser utilizado em qualquer lugar que count seria.

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

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

Nos exemplos acima mostra-se o funcionamento dos ponteiros. No primeiro, o código de formato %p usado na função printf( ) indica à função que ela deve exibir um endereço.

```
#include <stdio.h>
main( ) {
    int x,*ptrx;
    x = 1;
    ptrx = &x;
    printf("x = %d\n", x);
    printf("ponteiro do x, ptrx = %u\n", ptrx);
    printf("*ptrx+1 = %d\n", *ptrx+1);
    printf("ptrx = %u\n",ptrx);
    printf("*ptrx = %d\n",*ptrx);
    printf("*ptrx+=1 = %d\n",*ptrx+=1);
    printf("ptrx = %u\n",ptrx);
    printf("(*ptrx)++ = %d\n",(*ptrx)++);
    printf("ptrx = %u\n",ptrx);
    printf("*(ptrx++) = %d\n",*(ptrx++));
    printf("ptrx = %u\n",ptrx);
    printf("*ptrx++ = %d\n",*ptrx++);
    printf("ptrx = %u\n",ptrx);
}
```

Operadores aritméticos e de comparação com ponteiros

Com variáveis ponteiros é possível utilizar os operadores aritméticos de incremento, decremento, soma e subtração de inteiros. Mas estas operações não funcionam como com inteiros. Quando é incrementado um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, o endereço armazenado no ponteiro não é incrementado em um, ele é incrementado tanto quanto for preciso para apontar para a próxima variável contígua àquela que estava anteriormente apontando. Se o ponteiro apontava para um inteiro e é incrementado ele passa a apontar para o próximo inteiro; o endereço armazenado é incrementado em 2 bytes para pular o inteiro anteriormente apontado. Se apontava para um double o endereço será incrementado em 8 bytes. É uma

das razões porque o compilador precisa saber o tipo de um ponteiro. O decremento funciona analogamente.

É necessário identificar a diferença entre as seguintes instruções:

```
*(ptr++);
(*ptr)++;
```

No primeiro incrementa-se o ponteiro apontando para o próximo inteiro e retorna-se o conteúdo da variável agora apontada. No segundo caso, retorna-se o conteúdo da variável apontada por ptr e este conteúdo é incrementado em um.

Na soma ou subtração de inteiros com ponteiros, faz-se o ponteiro avançar ou retroceder tantas posições do tipo da variável quanto seja o resultado da operação aritmética. Se for soma acrescenta-se o endereço em bytes, se for subtração decrementa-se o endereço.

Exemplo:

```
ptr = ptr + 15;     /* ptr aponta 15 posições de inteiros após a posição antiga */
itens = *(ptr + 1);  /* atribui-se a itens o conteúdo do próximo inteiro em relação ao atual */
```

Às vezes é útil comparar dois ponteiros. Utilizando ptr1 == ptr2, e ptr1 != ptr2, se testa se dois ponteiros são ou não são iguais, isto é se estão apontando a mesma posição na memória. No caso de operações do tipo ptr1 > ptr2, ptr1 < ptr2, ptr1 >= ptr2 e ptr1 <= ptr2, compara-se qual ponteiro aponta para uma posição mais adiante na memória. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair números em ponto flutuante a ponteiros.

Ponteiros para ponteiros

Um ponteiro para um ponteiro é uma forma de indicação múltipla. Em um ponteiro normal, o valor do ponteiro é o valor do endereço da variável que contém o valor desejado. No caso de ponteiro para ponteiro o primeiro ponteiro contém o endereço do segundo, que aponta para a variável que contém o valor desejado. Exemplo : double **ptrdeptr;

O ptrdeptr é um ponteiro para um ponteiro que está apontando a uma variável do tipo double.

```
#include <stdio.h>
main( )
{
    int x,*p,**q;
    x=10;
    p=&x;
    q=&p;
    printf("%d",**q);
}
```

O erro chamado de *ponteiro perdido* é um dos mais difíceis de se encontrar, pois a cada vez que a operação com o ponteiro é utilizada, poderá estar sendo lido ou gravado em posições desconhecidas da memória. Isso pode acarretar em sobreposições sobre áreas de dados ou mesmo área do programa na memória. Exemplo:

```
int *p;
x=10;
*p=x;
```

Foi atribuído o valor 10 a uma localização desconhecida de memória. A conseqüência desta atribuição é imprevisível.

Nota : Muito cuidado no uso dos ponteiros. É importante você saber sempre para onde o ponteiro está apontando. Nunca utilize um ponteiro que não foi inicializado.

Ponteiros e vetores

Em C existe um grande relacionamento entre ponteiros e matrizes, sendo que eles podem ser tratados da mesma maneira. As versões com ponteiros geralmente são mais rápidas.

Vetores como ponteiros

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

tipo_da_variável nome_da_matriz [tam1][tam2] ... [tamN];

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é calculado multiplicando : tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo. O compilador então aloca este número de bytes em um espaço livre de memória. O nome da variável que você declarou é na verdade um ponteiro para o tipo da variável da matriz. Este conceito é fundamental, porque tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o primeiro elemento da matriz.

Então como é possível usar o seguinte? nome_da_matriz[índice]
Isto pode ser facilmente explicado desde que você entenda que a notação acima é absolutamente equivalente a se fazer:

*(nome_da_matriz + índice)

Assim funciona um vetor. Fica claro então, porque é que, no C, a indexação começa com zero. É porque, ao pegar o valor do primeiro elemento de um vetor, quer-se de fato o conteúdo do ponteiro nome_da_matriz, isto é *nome_da_matriz. Daí tem-se um índice igual a zero. Então

*nome_da_matriz é equivalente a nome_da_matriz[0].

Cuidado com os limites do vetor. C não verifica o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

Para varrer todos os elementos de uma matriz de uma forma seqüencial é bem mais simples utilizar ponteiros. Considere o seguinte programa para zerar uma matriz.

```
#include <stdio.h>
main ()
{
    float matrix [50][50];
    int i,j;
    for (i=0;i<50;i++)
        for (j=0;j<50;j++)
            matrix[i][j]=0.0;
}
```

Uma maneira muito mais eficiente será,

```
#include <stdio.h>
main ()
{
    float matrix [50][50];
    float *p;
    int count;
    p=matrix[0];
    for (count=0;count<2500;count++) {
        *p=0.0;
        p++;
    }
}
```

Você consegue ver porque o segundo programa é mais eficiente? Simplesmente porque cada vez que se faz matrix[i][j] o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.

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

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;
```

```
/* as operacoes a seguir sao invalidas */
    vetor = vetor + 2;     /* ERRADO: vetor nao e' variavel */
    vetor++;               /* ERRADO: vetor nao e' variavel */
    vetor = ponteiro;      /* ERRADO: vetor nao e' variavel */
```

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

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

Ponteiros como vetores

Lembre que o nome de um vetor é um ponteiro constante e que pode ser indexado. Identicamente todo ponteiro pode ser indexado, isto é, para um ponteiro ptr, a expressão ptr[3] equivale a *(p+3).

```
#include <stdio.h>
main ( ) {
    int values[] = { 11, 21, 31, 44, 55, 66, 77, 88, 99, 101 };
    int *p;
    p = values;
    printf ("Elemento 4 do vetor values,  values[3] = %d",p[3]);
}
```

Aproveitando o exposto sobre a indexação de ponteiros, a expressão :

&nome_do_ponteiro[índice]

é válida e retorna o endereço da casa do vetor indexado por índice. Como conseqüência, o ponteiro nome_do_ponteiro tem o endereço &nome_da_variável[0], o que indica onde na memória está guardado o valor do primeiro elemento do vetor.

Ponteiros e strings

Um caso interessante é a inicialização de ponteiros com strings constantes. Toda string inserida em um programa é colocada em um banco de strings que o compilador cria. No local onde está uma string no programa, o compilador coloca o endereço do início daquela string (que está no banco de strings). Como exemplo, se analisa a função strcpy(...) que pede dois parâmetros do tipo char*. Uma declaração da forma strcpy (string, "String

constante.") faz com que o compilador substitua a string "String constante." pelo seu endereço no banco de strings.

Isto sugere a inicialização de um ponteiro para uma string que vai ser utilizada várias vezes.

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

Então em todo lugar que se necessite a string pode se usar a variável str1. Observe que se o ponteiro é alterado é perdida a string. Por outro lado, se é utilizado o ponteiro para alterar a string pode-se facilmente corromper o banco de strings que o compilador criou.

Em resumo, nomes de strings são do tipo char*. Lembre que os ponteiros são poderosos mas, se usados com descuido podem ser uma fonte interminável de erros. Como exemplo, define-se uma nova função similar a strcpy(...).

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

StrCpy (char *destino, char *origem) {
    while (*origem) {
       *destino=*origem;
       origem++;
       destino++;
    }
    *destino='\0';
}

main ( )  {
    char str1[100],str2[100],str3[100];
    printf ("Entre com uma string: ");
    gets (str1);
    StrCpy (str2, str1);
    StrCpy (str3, "Voce digitou a string ");
    printf ("\n\n%s%s", str3, str2);
}
```

No programa acima, note que se pode passar ponteiros como argumentos de funções. Passando o ponteiro você possibilita à função alterar o conteúdo das strings. Você já fez isto anteriormente. No comando while (*origem) usa-se o fato de que a string termina com '\0' como critério de parada. Ao fazer origem++ e destino++ o leitor poderia argumentar que estamos alterando o valor do ponteiro base da string, mas como será explicado na próxima seção os ponteiros passados à função são apenas cópias, assim os ponteiros originais str1 e str2 permanecem inalterados na função main( ).

Ponteiros para ponteiros. Vetores de ponteiros

Podem ser construídos vetores de ponteiros como os vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser, int *vetorptr [10];. Assim vetorptr é um vetor que armazena 10 ponteiros para inteiros. Como existe uma certa equivalência entre vetores e ponteiros, diz-se que vetorptr é um ponteiro para ponteiro.

Um ponteiro para um ponteiro é como se você anotasse o lugar (endereço) de uma agenda que tem o endereço da casa do seu amigo. A sintaxe para declarar um ponteiro para um ponteiro é:

tipo_da_variável ****nome_da_variável**;

Se no código do programa utiliza-se a expressão : **nome_da_variável, está-se referenciando ao conteúdo final da variável apontada; enquanto, *nome_da_variável é o conteúdo do ponteiro intermediário.

Em conseqüência, no C pode-se declarar ponteiros para ponteiros para ponteiros, ponteiros para ponteiros para ponteiros para ponteiros, e assim por diante. Para fazer isto, basta aumentar o número de asteriscos na declaração. A lógica é a mesma, mas a utilidade?

Mais sobre funções

Funções são as estruturas que permitem ao programador separar seus programas em blocos. Se não as tivéssemos, os programas geralmente seriam muito compridos e ilegíveis. Para fazer programas grandes e complexos é melhor construí-los bloco a bloco. A sintaxe foi vista na seção de introdução às funções.

Lembre que uma função tem um tipo-de-retorno que é o tipo de variável que a função vai retornar. O tipo default é int. Os argumentos da função ou os parâmetros informam ao compilador quais serão as entradas da função. O corpo da função é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas.

O comando return

Para retornar um valor de uma função utiliza-se o comando **return**. A sua sintaxe é :
return *valor_de_retorno*; ou simplesmente,
return;

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

Uma função pode ter mais de uma declaração return, mas a função é terminada quando o programa chega à primeira declaração return.

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

Os valores retornados pelas funções podem ser aproveitados para fazer atribuições. Mas não se pode fazer: function(a,b) = x; /* Erro */.
Se você não fizer nada com o valor de retorno de uma função ele será descartado. Por exemplo, a função printf(...) retorna um inteiro que em exemplo nenhum foi utilizado, ele é descartado. No exemplo vemos o uso de mais de um return em uma função.

Protótipos de Funções

Até agora, as funções apresentadas nos exemplos estão fisicamente antes da função main( ). Isto foi feito porque ao compilar a função main( ) onde é chamada outra função, deve-se saber com antecedência quais são os tipos de retorno e quais são os parâmetros dessa função para gerar o código corretamente. Estando a outra função antes da função main( ) o compilador já a teria compilado e já saberia o seu formato.
Mas, muitas vezes o programa está espalhado por vários arquivos. Ou seja, são usadas funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?

A solução são os protótipos de funções. Protótipos são apenas declarações de funções. Isto é, aí é declarada uma função que se pode usar, indicando o tipo de retorno e os argumentos de entrada que se precisa. O compilador toma então conhecimento do formato daquela função antes de compilá-la.
Sintaxe:
tipo_de_retorno **nome_da_função** (declaração_de_argumentos);

Aqui o tipo-de-retorno, o nome-da-função e a declaração-de-argumentos são os mesmos que se pretende usar quando realmente seja dada a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis.

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

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

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

O tipo void

Void quer dizer vazio e é isto mesmo que o void é. Ele permite fazer funções que não retornam nada e funções que não têm parâmetros! O protótipo de uma função que não retorna nada será:

void *nome_da_função* (declaração_de_parâmetros);

Neste caso, não existe valor de retorno na declaração return, portanto ele não é necessário na função.

Para funções que não têm parâmetros:

tipo_de_retorno *nome_da_função* (**void**);

Se a função não tem parâmetros e não retornam nada:

void *nome_da_função* (**void**);

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

O compilador acha que a função main() deve retornar um inteiro quando nada é especificado. Isto pode ser interessante quando se precisa que o sistema operacional receba um valor de retorno da função main( ). **Convenção** : se o programa retornar zero, significa que ele terminou normalmente, e, se o programa retornar um valor diferente de zero, significa que o programa teve um término anormal.

Funções em arquivo cabeçalho. Escopo dos parâmetros.

É o momento de falar um pouco mais dos arquivos cabeçalho, incluídos sempre em todos os exemplos para poder utilizar funções pré-definidas. Eles não possuem os códigos completos das funções. Eles só contêm protótipos de funções, o que é suficiente. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto. O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem".

A linkagem é o instante em que todas as referências a funções cujos códigos não estão nos atuais arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas. É possível criar e incluir arquivos cabeçalho definidos pelo usuário. Isto será dado em um curso mais avançado.

Os parâmetros ou parâmetros formais de uma função são declarados como sendo as entradas de uma função, eles são cópias das variáveis passadas para a função. Os parâmetros formais existem independentemente das variáveis que foram passadas para a função. Não há motivo para se preocupar com o escopo deles. O parâmetro formal é como se fosse uma variável local da função. Isto é, a alteração do valor de um parâmetro formal não terá efeito na variável que foi passada à função.

Chamada por Valor e Chamada por Referência

Quando é chamada uma função os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é chamado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios.

```
#include <stdio.h>
float sqr (float num);

void main (void)
{
    float num,sq;
    printf ("Entre com um numero: ");
    scanf ("%f",&num);
    sq=sqr(num);
    printf ("\n\nO numero original e: %f\n",num);
    printf ("O seu quadrado vale: %f\n",sq);
}

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

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

Outro tipo de chamada de função é quando se precisa que as alterações dentro das funções nos parâmetros formais alterem os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim as suas referências, e então a função usa as referências para alterar os valores das variáveis fora da função.

O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem estar preocupados em estar mexendo nos valores dos

parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, por exemplo quando o novo valor calculado dentro da função deve ser priorizado. No C, existe um recurso de programação que pode ser usado para simular uma chamada por referência.

Para alterar as variáveis que são passadas para uma função, deve-se declarar seus parâmetros formais como sendo ponteiros. Os ponteiros são a "referência" que é necessária para poder alterar a variável fora da função. O único inconveniente é que, para chamar a função, deve-se lembrar de colocar um & na frente das variáveis que estiverem sendo passadas para a função. Como exemplo lembre da função scanf(...). Nela passavam-se os nomes das variáveis precedidos por &. Isto é, a função scanf(...) utiliza chamada por referência porque ela precisa alterar as variáveis que são passadas para ela.

```
#include <stdio.h>
void Swap (int *a,int *b);

void main (void)
{
    int num1,num2;
    num1=100;
    num2=200;
    Swap (&num1,&num2);
    printf ("\n\nEles agora valem %d  %d\n",num1,num2);
}

void Swap (int *a,int *b) {
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
```

A chamada por referência é muito utilizada para passar matrizes como argumentos. Por exemplo, seja a matriz : int matrix [50]; e seja a função function(...) que precisa trabalhar com os elementos da matriz, então existem três maneiras para declarar a função com argumento a matriz:

```
void function (int matrix[50]);
void function (int matrix[]);
void function (int *matrix);
```

Nos três casos, têm-se dentro de function(...) um int* chamado matrix. Isto significa que no caso de passar uma matriz para uma função, ela é passada através de um ponteiro. Então é possível alterar o valor dos elementos da matriz dentro da função.

Argumentos da função main

A função main() pode ter parâmetros formais. Mas o programador não pode escolher quais serão eles. A declaração mais completa que se pode ter para a função main( ) é:

int **main** (int **argc**, char ***argv[]**);

Os parâmetros argc e argv dão ao programador acesso à linha de comando com a qual o programa foi chamado.

O **argc** (argumento contador) é um inteiro e possui o número de argumentos com os quais a função main( ) foi chamada na linha de comando. Ele é no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.

O argv (argumento valores) é um ponteiro para uma matriz de strings. Cada string desta matriz é um dos parâmetros da linha de comando. O argv[0] sempre aponta para o nome do programa que é considerado o primeiro argumento. É para saber quantos elementos temos em argv que temos argc.

Recursividade

Na linguagem C assim como em muitas outras linguagens de programação, uma função pode chamar a si própria. Uma função assim é chamada função recursiva. Todo cuidado é pouco ao se fazer funções recursivas. A primeira coisa a se providenciar é um critério de parada. Este vai determinar quando a função deverá parar de chamar a si mesma. Isto impede que a função se chame infinitas vezes. Um bom exemplo é a função fatorial(int n).

```
#include <stdio.h>
int fatorial(int n)  {
    int ret;
    if (n)  return n*fatorial(n-1);
    else return 1;
}
void main( )  {
    int n;
    printf("\n\nDigite um valor para n: ");
    scanf("%d", &n);
    printf("\nO fatorial de %d e' %d", n, fatorial(n));
}
```

Um perigo ao utilizar funções recursivas é que a memória do computador pode ser esgotada rapidamente.

Ponteiros para funções

O C permite acessar variáveis e funções através de ponteiros. Esta é mais uma característica que mostra a força da linguagem de programação C. Pode-se então fazer coisas como, por exemplo, passar uma função como argumento para uma outra função. Sintaxe para o ponteiro para função.

tipo_de_retorno **(*nome_do_ponteiro)**( );
tipo_de_retorno **(*nome_do_ponteiro)**(declaração_de_parâmetros);

Veja que não é obrigatório declarar os parâmetros da função.

```
#include <stdio.h>
#include <string.h>
void PrintString (char *str,int (*fptr)());
main (void)  {
    char String [20]="Curso de C.";
    int (*p)();
    p=puts;
    PrintString (String,p);
    return 0;
}
void PrintString (char *str,int (*fptr)()) { (*fptr)(str); }
```

No programa acima, a função PrintString() usa uma função qualquer fptr para imprimir a string na tela. O programador pode então fornecer não só a string mas também a função que será usada para imprimí-la. No main( ) observa-se como pode ser atribuído ao ponteiro para funções p o endereço da função puts( ) do C.

Alocação dinâmica de memória

A alocação dinâmica permite ao programador criar variáveis em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado. Esta é outra ferramenta que mostra o poder do C. O padrão C ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca stdlib.h:

malloc
calloc
realloc
free

No entanto, existem diversas outras funções que são amplamente utilizadas, mas dependentes do ambiente e compilador. Neste curso são abordadas apenas as funções básicas mencionadas.

malloc

A função **malloc**( ) serve para alocar memória e tem o seguinte protótipo:

void ***malloc** (unsigned int num);

A função toma o número de bytes que se quer alocar (num), aloca na memória e retorna um ponteiro void * para o primeiro byte alocado. O ponteiro void * pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função malloc( ) retorna um ponteiro nulo.

```
#include <stdio.h>
#include <stdlib.h>
main (void)
{
    int *p;
    int n;
    ... /* Determina o valor de a em algum lugar */
    p=(int *)malloc(n*sizeof(int));
    if (!p) {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
    }
    ...
    return 0;
}
```

No exemplo acima, é alocada memória suficiente para se colocar n números inteiros. O operador sizeof( ) retorna o número de bytes de um inteiro. Ele é util para se saber o tamanho de tipos. O ponteiro void* que malloc( ) retorna é convertido para um int* pelo cast e é atribuído a p. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, p terá um valor nulo, o que fará com que !p retorne verdadeiro. Se a operação tiver sido bem sucedida, pode-se usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de p[0] a p[(a-1)].

calloc

A função **calloc**( ) também serve para alocar memória, mas possui um protótipo diferente:

void ***calloc** (unsigned int num, unsigned int size);

A funçao aloca uma quantidade de memória igual a num * size, isto é, aloca memória suficiente para uma matriz de num objetos de tamanho size. Retorna um ponteiro void * para o primeiro byte alocado. O ponteiro void * pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função calloc( ) retorna um ponteiro nulo.

```
#include <stdio.h>
#include <stdlib.h>
main (void)
{
    int *p;
    int a;
    ...

    p=(int *)calloc(a, sizeof(int));
    if (!p) {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
}
...
return 0;
}
```

realloc

A função realloc() serve para realocar memória e tem o seguinte protótipo:

void ***realloc** (void *ptr, unsigned int num);

A funçao modifica o tamanho da memória previamente alocada apontada por *ptr para aquele especificado por num. O valor de num pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque realloc() pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida. Se ptr for nulo, aloca size bytes e devolve um ponteiro; se size é zero, a memória apontada por ptr é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

free

Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária. Para isto existe a função free( ) cujo protótipo é:

void free (void *p);

Basta então passar para free( ) o ponteiro que aponta para o início da memória alocada. Mas você pode se perguntar, como é que o programa vai saber quantos bytes devem ser liberados? Ele sabe pois quando você alocou a memória, ele guardou o número de bytes alocados numa "tabela de alocação" interna.

```
#include <stdio.h>
#include <alloc.h>
main (void)  {
    int *p;
    int a;
    ...
    p=(int *)malloc(a*sizeof(int));
    if (!p) {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
    }
    ...
    free(p);
    ...
    return 0;
}
```

Alocação Dinâmica de Vetores

A alocação dinâmica de vetores utiliza os conceitos aprendidos na aula sobre ponteiros e as funções de alocação dinâmica apresentados. Um exemplo de implementação para vetor real:

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

float *Alocar_vetor_real (int n)  {
    float *v;        /* ponteiro para o vetor */
    if (n < 1) {  /* verifica parametros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (NULL);
    }
    /* aloca o vetor */
    v = (float *) calloc (n+1, sizeof(float));
    if (v == NULL)  {
        printf ("** Erro: Memoria Insuficiente **");
        return (NULL);
    }
    return (v);    /* retorna o ponteiro para o vetor */
}

float *Liberar_vetor_real (int n, float *v)  {
    if (v == NULL) return (NULL);
    if (n < 1) { /* verifica parametros recebidos */
        printf ("** Erro: Parametro invalido **\n");
```

```
        return (NULL);
    }
    free(v);        /* libera o vetor */
    return (NULL);  /* retorna o ponteiro */
}

void main (void)
{
    float *p;
     int a;
     ...    /* outros comandos, inclusive a inicializacao de a */
     p = Alocar_vetor_real (a);
     ...    /* outros comandos, utilizando p[] normalmente */
     p = Liberar_vetor_real (a, p);
}
```

Alocação Dinâmica de Matrizes

A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença de que se tem um ponteiro apontando para outro ponteiro que aponta para o valor final, o que é denominado indireção múltipla. A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro. Um exemplo de implementação para matriz real bidimensional é fornecido a seguir. A estrutura de dados utilizada neste exemplo é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz. Em cada linha existe um vetor alocado dinamicamente, como descrito anteriormente (compondo o segundo índice da matriz).

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

float **Alocar_matriz_real (int m, int n)
{
    float **v;  /* ponteiro para a matriz */
    int   i;    /* variavel auxiliar      */

    if (m < 1 || n < 1) { /* verifica parametros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (NULL);
    }

    /* aloca as linhas da matriz */
    v = (float **) calloc (m+1, sizeof(float *));

    if (v == NULL) {
```

```
        printf ("** Erro: Memoria Insuficiente **");
        return (NULL);
    }

    /* aloca as colunas da matriz */
    for ( i = 0; i <= m; i++ ) {
        v[i] = (float*) calloc (n+1, sizeof(float));

        if (v[i] == NULL) {
            printf ("** Erro: Memoria Insuficiente **");
            return (NULL);
        }
    }
    return (v); /* retorna o ponteiro para a matriz */
}

float **Liberar_matriz_real (int m, int n, float **v)
{
    int  i;  /* variavel auxiliar */
    if (v == NULL) return (NULL);

    if (m < 1 || n < 1) {  /* verifica parametros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (v);
    }

    for (i=0; i<=m; i++) free (v[i]); /* libera as linhas da matriz */
    free (v);      /* libera a matriz */
    return (NULL); /* retorna um ponteiro nulo */
}

void main (void)
{
    float **mat;  /* matriz a ser alocada */

    int   l, c;   /* numero de linhas e colunas da matriz */
    ...           /* outros comandos, inclusive inicializacao para l e c */

    mat = Alocar_matriz_real (l, c);
    ...           /* outros comandos utilizando mat[][] normalmente */

    mat = Liberar_matriz_real (l, c, mat);
    ...
}
```

Estrutura, união e enumeração

Estrutura

Uma estrutura agrupa várias variáveis numa só. Funciona como uma ficha pessoal, como registro de vários dados. O registro é a estrutura.

Para se criar uma estrutura utiliza-se a palavra-chave struct.
Sintaxe:

```
struct nome_do_tipo_da_estrutura
{
    tipo_1  nome_1;
    tipo_2  nome_2;
    ...
    tipo_n nome_n;
} variáveis_estrutura;
```

O nome_do_tipo_da_estrutura é o nome para a estrutura. As variáveis_estrutura são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que fazem parte do tipo nome_do_tipo_da_estrutura. Exemplo, uma estrutura para endereço, e outra para dados pessoais:

```
struct  tipo_endereco
{
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
};

struct  ficha_pessoal
{
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;
 };
```

Uma estrutura pode fazer parte de outra. Observe-se um exemplo onde são utilizadas as estruturas antes declaradas.

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

struct tipo_endereco
{
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
};

struct ficha_pessoal
{
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;
};

main (void) {
    struct ficha_pessoal ficha;
    strcpy (ficha.nome,"Luiz Osvaldo Silva");
    ficha.telefone=4921234;
    strcpy (ficha.endereco.rua,"Rua das Flores");
    ficha.endereco.numero=10;
    strcpy (ficha.endereco.bairro,"Cidade Velha");
    strcpy (ficha.endereco.cidade,"Belo Horizonte");
    strcpy (ficha.endereco.sigla_estado,"MG");
    ficha.endereco.CEP=31340230;
    return 0;
}
```

No exemplo mostra-se como acessar um elemento de uma estrutura. Basta usar o operador **.** ponto.

Matrizes de estruturas

Um estrutura é como qualquer outro tipo de dado no C. Podem ser declaradas, portanto, matrizes de estruturas da seguinte forma.

struct *ficha_pessoal* **fichas** [100];

Isto criará um vetor de 100 estruturas do tipo ficha_pessoal. Para acessar a segunda letra da sigla de estado da décima terceira ficha faz-se:

fichas[12].endereco.sigla_estado[1];

Atribuindo estruturas

Pode-se atribuir duas estruturas que sejam do mesmo tipo. O C irá, neste caso, copiar uma estrutura na outra. Esta operação não apresenta problemas pois ao declarar

struct ficha_pessoal ficha;

ficha não é um ponteiro, mas uma estrutura.

```
void main()
{
    struct ficha_pessoal primeira, segunda;
    Le_dados(&primeira);
    segunda = primeira;
    Imprime_dados(segunda);
}
```

São declaradas duas estruturas do tipo ficha_pessoal, uma chamada primeira e outra chamada segunda. Supondo que haja declarada uma função Le_dados() que faça a leitura de uma estrutura, admitimos que após a execução da Segunda linha de main(), a estrutura primeira estará preenchida com dados válidos. Os valores de primeira são copiados na segunda apenas com a expressão de atribuição.

Todos os campos de primeira serão copiados na ficha chamada segunda. Tomar cuidado com a seguinte declaração:

struct ficha_pessoal fichas [100];

pois neste caso fichas é um ponteiro para a primeira ficha. Para a estrutura completa da n-ésima ficha usar fichas[n-1];.

Estruturas como argumentos de funções

Em um exemplo acima, utilizou-se o seguinte comando:

strcpy (ficha.nome,"Luiz Osvaldo Silva");

Neste comando um elemento de uma estrutura é passado para uma função. Este tipo de operação pode ser feita sem maiores considerações.

Pode-se também passar para uma função uma estrutura inteira, isto da seguinte maneira.

```
void PreencheFicha (struct ficha_pessoal ficha)
{
    ...
}
```

É fácil passar a estrutura como um todo para a função. Observar que, como em qualquer outra função no C, a passagem da estrutura é feita por valor. Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função. Mais uma vez

podemos contornar este pormenor usando ponteiros e passando para a função um ponteiro para a estrutura.

Ponteiros para estruturas

Os ponteiros também podem ser utilizados para estruturas. A declaração de um ponteiro para uma estrutura é:

struct *nome_do_ponteiro_para_estrutura* *estptr;

Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados no C. Há um detalhe especial a ser considerado. Se o ponteiro estptr está apontando para uma estrutura e tenta acessar um elemento ou dado da estrutura deve-se utilizar a sintaxe:

(*p).nome_do_dado_da_estrutura;

Este formato raramente é usado. O que é comum de se fazer é acessar o elemento ou dado da estrutura através do operador seta (->). Assim, o anterior é equivalente a:

p->nome_do_dado_da_estrutura;

A declaração acima é muito mais fácil e concisa. Como exemplo, para acessar o dado CEP dentro de endereço, para um ponteiro a uma estrutura do tipo ficha_pessoal, faz-se:

p->endereco.CEP

União

Uma declaração **union** determina uma única localização de memória onde podem estar armazenadas várias variáveis diferentes. A declaração de uma união é semelhante à declaração de uma estrutura:

```
union nome_do_tipo_da_union
{
    tipo_1 nome_1;
    tipo_2 nome_2;
    ...
    tipo_n nome_n;
} variáveis_union;
```

Como exemplo, considerar a seguinte união:

```
union angulo
{
    float graus;
    float radianos;
};
```

Nela, existem duas variáveis (graus e radianos) que, apesar de terem nomes diferentes, ocupam o mesmo local da memória. Isto quer dizer que só gastamos o espaço equivalente a um único float. Uniões podem ser feitas também com variáveis de diferentes tipos. Neste caso, a memória alocada corresponde ao tamanho da maior variável no union.
Veja o exemplo:

```
#include <stdio.h>
#define GRAUS 'G'
#define RAD 'R'
union angulo  {
    int graus;
    float radianos;
};
void main()  {
    union angulo ang;
    char op;
    printf("\nNumeros em graus ou radianos? ");
    scanf("%c",&op);
    if (op == GRAUS) {
        ang.graus = 180;
        printf("\nAngulo: %d\n",ang.graus);
    }
    else if (op == RAD) {
        ang.radianos = 3.1415;
        printf("\nAngulo: %f\n",ang.radianos);
    }
    else printf("\nEntrada invalida!!\n");
}
```

Tomar o maior cuidado possível ao trabalhar com uniões, pois poderia ser feito o seguinte:

```
#include <stdio.h>

union numero  {
    char Ch;
    int I;
    float F;
};
main (void) {
    union numero N;
    N.graus = 123;
    printf ("%f",N.F);
    return 0;
}
```

O programa acima é muito perigoso pois você está lendo uma região da memória, que foi "gravada" como um inteiro, como se fosse um ponto flutuante.

Enumerações

Numa enumeração diz-se ao compilador quais os valores(constantes) que uma determinada variável pode assumir. Sintaxe:

enum *nome_do_tipo_da_enumeração* {lista_de_valores} lista_de_variáveis;

Exemplo:
enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado, domingo};

O programador diz ao compilador que qualquer variável do tipo dias_da_semana só pode ter os valores enumerados. Isto quer dizer que poderíamos fazer o seguinte programa:

```
#include <stdio.h>
enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado, domingo};
main (void)  {
    enum dias_da_semana d1,d2;
    d1=segunda;
    d2=sexta;
    if (d1==d2) {
        printf ("O dia e o mesmo.");
    }
    else {
        printf ("São dias diferentes.");
    }
    return 0;
}
```

O funcionamento da enumeração é simples, o compilador pega a lista de elementos declarada com a palavra enum, e associa a cada um destes elementos, um número inteiro. Ao primeiro elemento da lista é associado o número zero, ao segundo o número 1 e assim por diante. Isto significa que os elementos declarados em uma enumeração são todos inteiros.

Pode-se também modificar o valor inteiro destes elementos atribuindo valores inteiros específicos. Exemplo: enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado=-1, domingo};

Neste caso, o valor inteiro de sábado é –1. Tenha presente que o valor de um elemento consecutivo a um outro elemento dado, é incrementado em uma unidade. Assim sendo, o domingo tem valor zero, ao igual que o elemento segunda. Nota: Vários elementos de uma enumeração podem assumir o mesmo valor numérico inteiro.

Referências

1. *C, completo e total*. Herbert Schildt. Tradução Marcos R. A. Morais, Makron, Mc Graw-Hill. São Paulo. 1990.

2. *Introducción al lenguaje C*. Les Hancock, Morris Krieger. Traducción Sebastián Dormido B. Mc Graw-Hill. España. 1988.

3. *C Reference Manual*. Samuel p. Harbison, Guy L. Steele. Prentice-Hall Inc. Englewood, New Jersey. 1994.

4. *A linguagem de programação padrão ANSI*. Kernighan, B. & Ritchie, D. C. Editora Campus, 1990.

5. *Treinamento em linguagem C - Módulos 1 e 2*. Mizrahi, V. V. Editora McGraw-Hill, 1990.

Um pouco da historia nos Laboratórios Bell

O final da década dos 60 foi uma época turbulenta para a pesquisa de sistemas de computadores nos Laboratórios Telefônicos Bell. Esta companhia estava saindo do projeto Multics, que iniciou em união com as empresas MIT e General Eletric. Por volta de 1969, os administradores dos Laboratórios Bell e os pesquisadores, perceberam que o projeto Multics renderia frutos tarde demais e que isto custaria muito dinheiro. Depois que a máquina Multics GE-645 foi removida das premissas, um grupo informal liderado por Ken Thompson iniciou investigações alternativas.

Thompson pretendia criar um ambiente de computador confortável de acordo com seus projetos, usando todos os métodos que dispunha. Seus planos incorporavam muitas das inovações do projeto Multics, como:

- um sistema de arquivos estruturado em árvore,
- um interpretador de comandos trabalhando como um programa em nível de usuário,
- uma representação simples de arquivos texto,
- acesso generalizado a dispositivos, etc.

Foi usada a linguagem PL/I, utilizada no desenvolvimento do projeto Multics, e ela não satisfazia as necessidades. Foram usadas também outras linguagens, como BCPL, mas todas elas perdiam as vantagens dos programas escritos em nível de assembly.

BCPL foi escrito por Martin Richards no meio dos anos 60, enquanto ele estava visitando o MIT, e foi usado durante o início dos anos 70 em vários projetos interessantes, como o sistema operacional OS6 em Oxford, e parte de um projeto da Xerox. O compilador original foi transportado para o Multics e para o sistema GE-635 GECOS por Rudd Canaday e outros dos Laboratórios Bell.

Não muito tempo depois do primeiro Unix rodar em um PDP-7, em 1969, Doug McIlroy criou a primeira linguagem de alto nível do sistema: uma implementação da linguagem TMG (TMG era uma linguagem usada para escrever compiladores) de McClure.

Desafiado pelo feito por McIlroy na reprodução do TMG, Thompson decidiu que o Unix precisaria de uma linguagem de programação. Depois de uma rápida tentativa em Fortran, ele criou sua própria linguagem, que ele chamou de B. B pode ser imaginado como uma linguagem C, mas sem tipos; mais exatamente, era um BCPL reduzido em 8KB de memória, e filtrado pelo cérebro de Thompson. O nome B, mais provavelmente, representa uma contração do BCPL, embora haja uma teoria que diga que foi derivado da linguagem Bon, uma linguagem não registrada que Thompson criou durante os dias do Multics. Ao final da vida útil do Multics nos Laboratórios Bell e depois, B foi a linguagem de escolha entre o grupo de pessoas que mais tarde envolveram-se com o Unix.

O compilador B no PDP-7 não gerava instruções de máquina, mas sim "código em linha", um esquema interpretativo no qual a saída do compilador consiste numa seqüência de endereços de fragmentos de código que realizam operações elementares. As operações agem sobre a pilha da máquina.

Certamente os aspectos menos agradáveis do BCPL foram devidos aos próprios problemas tecnológicos, que foram anulados no projeto da linguagem B. Por exemplo, o BCPL usa um mecanismo de "vetor global" para a comunicação entre programas compilados separadamente. Neste esquema, o programador associa explicitamente o nome de cada procedimento visível externamente e os objetos de dados com um offset numérico no vetor global; a link-edição é

completada no código compilado usando estes offset's numéricos. B acabou com este inconveniente inicialmente insistindo em que o programa todo seja apresentado de uma vez ao compilador. Ambas as linguagens têm somente um tipo de dado, o 'word' ou 'célula', de tamanho fixo. A memória, nestas linguagens, consiste num vetor linear de muitas células, e o significado do conteúdo de uma célula depende da operação aplicada.

Depois que a versão TMG do B estava funcionando, Thompson reescreveu o B pelo próprio B. Durante o desenvolvimento, ele lutou continuamente contra as limitações de memória: cada linguagem adicional inchava o compilador, mas cada reescrita levava a vantagem da redução de tamanho. Thompson inventou os operadores ++ e --, que incrementam ou decrementam.

Em 1971 foi adicionado à linguagem B o tipo caracter e o seu compilador foi reescrito para gerar códigos de máquina ao invés de códigos de linha. Assim a transição de B para C foi na mesma época da criação de um compilador capaz de produzir programas rápidos e pequenos o suficiente para competir com a linguagem assembly. A nova linguagem foi chamada de NB inicialmente. Esta transição do B para o C foi feita pelo Dennis Ritchie em 1972.

Posteriormente, para a nomeação da nova linguagem, decidiu-se seguir o estilo da nomenclatura da linguagem anterior, com apenas uma letra, chamando-se de C, e ficando aberta uma questão: se o nome é apenas um incremento “no alfabeto” ou nas letras “BCPL”.

BCPL, B e C estão todos firmemente habilitados na tradicional família procedural tipificada por Fortran e Algol 60. Eles são particularmente orientados para sistemas de programação, são menores e compactamente descritos, e são submetidos a transformação por simples compiladores.

O esquema de composição de tipos do C deve muito ao Algol 68, embora ele não tivesse surgido de tal forma que os adeptos do Algol pudessem aproveitá-lo. A idéia central que foi capturada do Algol foi uma estrutura baseada em tipos atômicos (incluindo estruturas), compostos de vetores, ponteiros e funções.

Nem o BCPL, nem o B, e nem o C manipulam dados caracter robustamente. Cada uma destas linguagens trata uma string como um vetor de inteiros. Em BCPL, o primeiro pacote de byte contém o número de caracteres na string; em B, não há contador e as strings são terminadas por um caracter especial, assim como em C.

Por volta de 1982 ficou claro que o C precisava de uma padronização. A melhor aproximação a este padrão, a primeira edição de "Kernighan & Ritchie", descrevia a linguagem em uso atual. Mas isto foi insuficientemente preciso no que dizia respeito a certos detalhes da linguagem, deixando pouco divulgadas certas extensões. Para resolver este problema, ANSI estabeleceu no verão de 1983 o comitê X3J11, sob a direção de CBEMA, com a meta de produzir uma linguagem C padrão. O X3J11 emitiu o seu relatório no fim de 1989, e este padrão foi aceito pelo ISO como ISO/IEC 9899/1990 e é conhecido pelo nome de ANSI C.