UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO

INTRODUÇÃO À PROGRAMAÇÃO

Linguagem de Programação C

Roberto Medeiros de Faria

Agosto/2002

ÍNDICE

INTRODUÇÃO

C é uma linguagem de programação desenvolvida nos Laboratórios Bell, por volta de 1972, tendo sido projetada e escrita por Dennis Ritchie, que trabalhava junto com Ken Thompson no sistema operacional UNIX. Esse sistema foi concebido como um conjunto de ferramentas para engenheiros de software, das quais C se tornou a mais básica. Quase todos os utilitários do UNIX, seu núcleo e o compilador C são escritos em C.

A grande força de C é a habilidade para construir programas complexos como elementos simples, tanto que seu lema poderia ser "muito com pouco".

O C é destinado a construção de software básico, ou seja, linguagens de programação, sistemas operacionais, geradores de programas, processadores de texto, etc.

A linguagem C pode ser considerada uma linguagem de médio nível. O C oferece recursos de baixo nível permitindo a especificação de todos os detalhes na lógica de um programa, alcançando a máxima eficiência computacional, como também possui um relativo alto nível para abstrair os detalhes da arquitetura do computador.

REGRAS E COMANDOS DA LINGUAGEM C

Principais Extensões

Os tipos arquivos mais comuns envolvidos na edição, compilação, linkedição e execução de um programa C, têm as seguintes terminações padrão:

`.C` - Arquivo contendo o Código Fonte do programa representado por uma seqüência de caracteres ASCII;

`.H` - Arquivos cabeçalhos (header). Definições em código fonte para o programa e Bibliotecas pré-compiladas.

`.OBJ` - Arquivo objeto gerado a partir da compilação do código fonte.

`.EXE` - Código executável. Resultado da junção de um ou mais arquivos objetos e bibliotecas padrões pelo processo de linkedição.

Bloco de Código (Marcadores de Bloco – `{}`)

Um bloco de código é um grupo de comandos de programa, conectados logicamente, que o compilador trata como uma unidade. A sintaxe da linguagem C para crição de um bloco é uma seqüência de comando entre chaves. As chaves são usadas para delimitar um comando composto ou bloco de comandos - equivalente ao *begin ... end* da linguagem Pascal.

Ponto-e-Vírgula (;)

O ponto-e-vírgula é um terminador de comando e indica o final de um comando.

Comentários (`/* comentário */`)

Tudo que estiver entre /* e */ é considerado um comentário e será ignorado pelo compilador. Os comentários não interferem na execução do programa. Os comentários podem aparecer em qualquer parte do programa.

Identificadores

A linguagem C define identificadores como sendo nomes usados para se fazer referência a entidades do programa (variáveis, funções, rótulos, etc.) definidas pelo programador. Em C, um identificador é composto de um ou mais caracteres, sendo que, para identificadores internos, os 31 primeiros são significativos. O primeiro caractere deve ser uma letra ou um sublinha (_) e os caracteres subseqüentes devem ser letras, números ou sublinhas. Alguns compiladores C também permitem que você use um cifrão ($) dentro de um identificador, mas não como o caractere inicial. Eis aqui alguns exemplos de identificadores corretos e incorretos:

Corretos	**Incorretos**
`cont`	`1cont`
`valor23`	`alô`
`total_geral`	`total..geral`
`totalGeral`	`valor-total`

Isto quer dizer que se duas variáveis têm em comum os 31 primeiros caracteres e diferem apenas a partir do trigésimo segundo, o compilador C não será capaz de distingui-las. Por exemplo, esses dois identificadores são iguais:

```
isto_e_um_exemplo_de_um_nome_longo
isto_e_um_exemplo_de_um_nome_longo_também
```

Uma outra observação é que em C as letras maiúsculas e minúsculas são tratadas de maneira diferentes. Por exemplo, cont, Cont e CONT são três identificadores distintos.

Um identificador não pode ser igual a uma palavra reservada do C e não deve ter o mesmo nome das funções que você escreveu ou daquelas que estão pré-definidas nas bibliotecas do C.

Quando um identificador for composto por várias palavras, deve-se separa-las com sublinha ou iniciar cada palavra, a partir da segunda, com letra maiúscula.

Exemplos:

```
valor_convertido_para_celcius
valorConvertidoParaCelcius
```

Tipos de Dados

Existem cinco tipos primitivos de dados em C: caractere, inteiro, ponto flutuante, ponto flutuante de precisão dupla e sem valor. As palavras reservadas usadas para declarar variáveis desse tipo são char, int, float, double, void, respectivamente. A tabela abaixo representa o valor e a escala de cada tipo de dado em C.

Tipo	***Blts***	***Valor***
`char`	8	-128 a 127
`Int`	16	-32768 a 32767
`float`	32	3.4E -38 a 3.4E+38
`doublé`	64	1.7E-308 a 1.7E+308
`void`	0	sem valor

Modificadores de Tipos

Com exceção do tipo *void*, os tipos primitivos podem ser precedidos por modificadores. O modificador é usado para alterar o significado do tipo-base para que ele

se adapte de maneira mais precisa às necessidades das várias situações. Os modificadores de tipo são:

```
signed
unsigned
long
short
```

Abaixo estão todas as possibilidades para relacionar os tipos primitivos com os modificadores de tipo. Embora permitido, o uso de *signed* com Inteiros é redundante. porque a declaraçâo “à revelia” (default) dos inteiros assume um número com sinal.

Tipo	***Bits***	***Valor***
`Char`	8	-128 a 127
`unsigned char`	8	o a 255
`signed char`	8	-128 a 127
`Int`	16	-32768 a 32767
`unsigned int`	16	0 a 65535
`signed int`	16	-32768 a 32767
`short int`	16	-32768 a 32767
`unsigned short int`	16	0 a 65535
`signed short int`	16	-32768 a 32767
`long int`	32	-2147483648 a 2147483647
`signed long int`	32	-2147483648 a 2147483647
`unsigned long int`	32	0 a 4294967295
`Float`	32	3.4E-38 a 3.4E+38
`Doublé`	64	1.7E-308 a 1.7E+308
`long double`	90	3.4E-4932 a 1.1 E+4932
`Void`	0	sem valor

A diferença entre inteiros com sinal e inteiros sem sinal está na maneira com que o computador interpreta o bit de ordem superior (mais à esquerda) do inteiro. Se você especificar um inteiro com sinal, então o compilador do C gerará um código que assumirá que ele deve usar o bit de ordem superior de um inteiro para representar o sinal. Se o bit de sinal é zero, então o número é positivo; se o bit de sinal é um, então o número é negativo. Para o inteiro sem sinal, o bit de maior ordem representa um dígito binário como os demais e não há representação de sinal o valor é sempre positivo.

Definição de Variáveis

Definir uma variável é cria-la na memória (aloca-la), dar a ela um nome e especificar o tipo de dado que nela vai armazenar.

Sintaxe:

```
tipo lista-de-identificadores;
```

O tipo deve ser um tipo de dado válido em C, e lista-de-identificadores pode consistir em um ou mais identificadores separados por vírgula. Exemplos:

```
int soma ;
unsigned char i,j,k ;
float salário;
signed long [int] x;
unsigned int idade;
short int y;
long caminho, estado;
unsigned valor;
```

Atribuição e Inicialização de Variáveis

O símbolo usado como operador para atribuir o valor de uma expressão a uma variável é o igual (=). Exemplos:

```
soma = 10;
delta = b * b – 4 * a * c;
```

A atribuição também pode ser feita diretamente na declaração. Exemplos:

```
int soma = 10;
char i = 3 + 2;
```

Operadores

Relacionais:

```
==   (igual a);
!=   (diferente de);
>    (maior do que);
<    (menor do que);
>=   (maior ou igual a);
<=   (menor ou igual a).
```

Aritméticos:

```
+    (soma);
-    (subtração);
*    (multiplicação);
/    (divisão);
%    (resto da divisão inteira).
++   (incremento)
--   (decremento)
```

Lógicos:

```
!    (negação – not – não);
||   (disjunção – or – ou);
```

`&&  (conjunção – and – e).`

Incremento e Decremento

Na linguagem C existem operadores especiais para as operações de incremento (adicionar um) e decremento (subtrair um) de variáveis.

Suponha que uma variável y deva ser incrementada de uma unidade antes de seu valor ser atribuído à variável x, temos então o seguinte trecho de programa:

```
y = y + 1;
x = y;
```

Usando o operador de incremento pré-fixado, o mesmo trecho do programa acima pode ser reduzido a:

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

Com a operação de incremento pós-fixado, temos:

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

Dessa formal y é atribuído a x para depois ser incrementado, ou seja:

```
x = y;
y = y + 1;
```

O mesmo raciocínio se aplica quando se trata de uma operação de decremento. O operador de decremento subtrai um do valor de uma variável. O operador de decremento pode ser também, pré-fixado e pós-fixado.

Expressões de Atribuição

É usada para substituir trechos de programas, como por exemplo:

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

Usando expressão de atribuição teríamos o mesmo trecho, escrito como:

```
x += y;
```

Ou seja, quando a variável a que se atribuirá o valor da expressão é igual ao primeiro operando e na expressão só se tem um único operador, pode-se utilizar expressões e atribuição.

Esta forma se aplica as quatro operações elementares e ao resto da divisão inteira (`+=, -=, *=, /=, %=`).

Atribuições Múltiplas

Usadas para na iniclalizar várias variáveis com um único valor de uma expressão. Exemplo:

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

É importante observar que o valor zero é atribuído às variáveis em ordem da direita para à esquerda. Então, a mesma expressão, com parênteses (para indicar a associatividade) seria:

```
x = (y = (z = 0));
```

Ou seja, seria o mesmo que escrever as atribuições separadamente:

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

Prioridade dos Operadores

É multo importante que saibamos a precedência e associatividade dos operadores, para que uma expressão seja avaliada e tenha o resultado esperado. A tabela abaixo lista todos os operadores que iremos trabalhar. Alguns, você já conhece.

OPERADOR	*NÍVEL*	*NOME*
[ ]	1	arrav
( )		função
->		membro
.		membro
!	2	negação
~		complemento
++		incremento
--		decremento
-		menos unário
(tipo)		Cast
*		indirecão
&		endereço
Sizeof		tamanho do objeto
*	3	multiplicação
/		divisão
%		resto da divisão inteira
+		adição
-		subtração
<<	4	deslocamento à esquerda

>>		deslocamento à direita
<	5	menor que
<=		menor ou igual a
>		maior que
>=		maior ou igual a
==	6	igual a
!=		diferente de
&	7	e para bits
^	8	ou para bits (exclusivo)
|	9	ou para bits
&&	10	e lógico
||	11	ou lógico
?	12	expressão condicional
=	13	atribuicão
,	14	multipla expressão

Notação Hexadecimal e Octal

Dependendo do tipo de aplicação que estamos trabalhando, as vezes, é mais cômodo usar a notação hexadecimal ou a octal.

Para indicarmos que um valor está escrito em hexadecimal, devemos iniciá-lo com 0x.

Exemplo:

```
vbit = 0xFA14;
flag = 0xFF;
```

Para valores octais basta iniciar com 0.

Exemplo:

```
voct = 0123;
masc = 088;
```

Note que 09 é um número em octal e portanto é diferente de 9 em decimal.

Constante Caractere

Uma constante caractere em C é delimitada por apóstrofo (').

Exemplo:

```
'a', 'b', '1', '0' , ‘9’
```

Constante de Caracteres Especiais

As constantes de caracteres especiais são representadas da mesma forma que as constantes de caracteres comuns, porém, caso o caractere não tenha representação visual ou seu uso confunda a verificação sintática da linguagem C, usa-se o caractere contra-barra (barra invertida) para auxiliar na sua representação. A tabela abaixo ilustra alguns dos usos do caractere contra-barra ('\') na representação na representação de caracteres especiais.

SEQÜÊNCIA	*VALOR*	*SIGNIFICADO*
'\0'	0	finalizador de um string(NULL)
'\8'	0x07	apito sonoro (bell)
'\b'	0x08	retrocesso (backspace)
'\f'	0x0C	avanço para o início de outra folha
'\n'	0x0A	avanço de linha (new line)
'\r'	0x0D	retorno do cursor (return)
\t'	0x09	tabulação horizontal (tab)
'\v'	0x0B	tabulação vertical
'\\'	0x5C	barra invertida (contra-barra)
'\''	0x27	apóstrofo
'\"'	0x22	aspas

Constante Cadeia de Caracteres (String)

Toda seqüência de caracteres que estiver entre aspas (").

Exemplos:

```
"Empresa de Demonstração", "123.456"
```

Conversão de Tipo

Quando operadores de tipos diferentes aparecem em expressões aritméticas, eles são convertidos para um tipo comum (default), de acordo com as seguintes conversões:

char e short são convertidos para int;

float é convertido para double.

Argumentos passados para uma função também estão sujeitos a regras de conversão.

Há ainda a possibilidade de conversão forçada (cast) em uma expressão, com a construção:

```
(tipo)expressão;
```

Esta construção converte a expressão para o tipo "tipo". Por exemplo, a função sqrt() da biblioteca C, espera sempre um argumento em precisão dupla assim se tivermos uma variável numero do tipo inteiro sua raiz deve ser calculada como:

```
raiz =sqrt((double)numero);
```

Exemplo:

```
char i;
unsigned char k;
int j;
i = (char)j;
k = (unsigned char)i;
j = (unsigned char)calcxy();  /* converte o retorno da função
                                 */
```

SAÍDA DE DADOS

Sintaxe:

```
printf("cadeia de controle", lista-de-argumentos) ;
```

A “cadeia de controle” especifica o formato de saída da lista-de-argumentos. A tabela abaixo mostra os códigos que podem ser usados na cadeia de controle como especificação de formato:

CÓDIGO	*FORMATAÇÃO*
%c	caractere
%d	decimal
%f	ponto flutuante
%u	decimal sem sinal
%x	hexadecimal
%o	octal
%p	apontador
%s	Cadeia de caracteres (string)
%e	notação científica (exponencial)

Os códigos de formatos poderão ter especificações da largura do campo, número de casas decimais e o indicador de alinhamento à esquerda. Um inteiro colocado entre o sinal (%) e o comando de formatação atua como especificador da largura de campo mínima, o que preenche a saída com brancos ou zeros para assegurar que tenha ao menos um campo mínimo. Caso uma série ou um número sejam maiores que o mínimo, serão completamente impresso, mesmo que ultrapasse o mínimo. O preenchimento normal é feito com espaço, Caso você queira preencher com zeros, basta colocar um zero antes do especificador de largura de campo. Por exemplo %05d vai preencher um número como menos de cinco dígito, com 0's a esquerda.

Para especificar o número de casas decimais impressas para um número em pontos flutuante. coloque um ponto decimal seguido pelo número de casas decimais que você quer apresentar, após o especificador de largura de campo. Por exemplo %10.4f apresentará um número com pelo menos 10 caracteres de comprimento, com quatro casas decimais. Esse método também funciona quando você quer especificar o comprimento máxImo do campo, para string e valores inteiros. Por exemplo, %5.75 apresentara um string com pelo menos cinco caracteres de comprimento e que não exceda sete. Caso o string seja mais comprido que a máxima largura do campo. os caracteres serão truncados na extremidade final.

Como condição normal, toda sarda é alinhada pela direita: se a largura do campo for mais larga que os dados impresso, os dados serão colocados no lado direito do campo. Você pode formatar a informação a ser colocada no lado esquerdo, pela colocação de um sinal de menos logo após %. Por exemplo, %-10.2f fará um alinhamento à esquerda.

Exemplos:

```
printf( "Empresa de Demonstração” ) ;
printf( "%s", "Empresa de Demonstração”);
printf( “%c", ‘B’);
printf("%5d", 100);
printf("%-05d", 100);
```

ENTRADAS DE DADOS

Sintaxe:

```
scanf("cadeia de controle", <lista de argumentos>)
```

A cadeia de controle especifica o tipo de dados que se deseja ler do teclado. A tabela abaixo mostra os códigos que podem ser usados na cadeia de controle:

CÓDIGO	*SIGNIFICADO*
%c	ler um caractere
%d	ler um inteiro
%f	ler um número em ponto flutuante
%x	ler um número em hexadecimal
%o	ler um número em octal
%s	ler uma cadeia de caracteres (string)

Todos os argumentos devem ser precedido do sinal &. Por exemplo se você quiser ler um inteiro para a variável inteira cont, deverá utilizar o seguinte comando:

```
scanf("%d", &cont);
```

Os comandos de formatação podem utilizar modificadores de largura de campo, formados por números inteiros colocados entre o sinal % e o código de formatação. Se for colocado um *, a função scanf() avançará para o próximo campo de entrada.

Exemplos:

```
scanf("%c", &op);
scanf("%30s", &nome);
scanf("%d", &idade);
scanf("%d %c", &ld, &ch);
```

CONTROLE DE FLUXO

Comando `if`

Sintaxe:

```
if(condição)
     comando1;
else
     comando2;
```

Caso a condição seja verdadeira, ou seja, o valor da expressão condicional seja diferente de zero, o comando1 será executado; caso contrário, será executado o comando2.

A cláusula *else* é opcional.

Os comandos, tanto no if quanto no *else*, podem ser comandos de blocos.

Exemplo:

```
void main() {
     int valorA, valorB;
     scanf("%d %d", &valorA, &valorB);
     if(valorA > valorB) {
          printf("%d ", valorA);
     }
     else {
          printf("%d ", valorB);
     }
}
```

Expressão Condicional (forma compacta para o `if`)

Existe uma forma mais prática, porém menos legível, para uma expressão condicional. Essa forma é bastante usada na definição de macros.

Sintaxe:

```
identificador = condição ? exp1 : exp2;
```

Exemplo:

```
max = a > b ? a : b;
```

A expressão acima é equivalente a:

```
if(a > b) {
     max = a;
```

```
}
else {
     max = b ;
}
```

Comando `switch`

Sintaxe:

```
switch(expressão-inteira) {
     case constante-inteira-1:
          comando1;
          break;
     case constante-inteira-2:
          comando2;
          break;
     default:
          comandoN;
     }
```

O comandos break e a cláusula default são opcionais.

O switch executará o primeiro bloco de comando em que a variável coincidir com a constante até encontrar um break. Caso nenhuma constante seja igual ao valor da variável a cláusula default será avaliada.

Exemplo:

```
switch(ch) {
     case '1':
          printf("opção um escolhida");
          break;
     case '2':
          printf("opção dois escolhida");
               break;
     case '3':
     case '4':
          printf("opção escolhida foi três ou quatro");
     default:
          printf("escolha um, dois, três ou quatro");
}
```

Comando `while`

Sintaxe:

```
while(condição)
     comando;
```

O comando pode ser um única declaração ou um bloco de comandos que devem ser repetidos. A condição pode ser qualquer expressão em que a verdade é qualquer valor diferente de zero.

Exemplo:

```
void main() {
     int valor = 10;
     while(valor >= 10) {
          printf("%d", valor);
     }
}
```

Comando `for`

Dos comandos vistos até agora, o comando for é o mais completo. Sua sintaxe e semântica difere, e muito, das outras linguagens.

Sintaxe:

```
for([lista-de-exp1]; [exp2]; [lista-de-exp3>])
     comando;
```

Onde:

lista de exp1 - é uma lista de expressões de inicialização;

exp2 – é uma expressão condicional;

lista de exp3 – é uma lista de expressões de reinicialização.

Semântica:

1º.) as expressões de Inlcialização são executadas.

2º.) a expressão condicional é avaliada; se for zero (falsa), o laço terminará; se for diferente de zero (verdadeira), os comandos internos serão executados, e em seguida, é executada a lista de expressões de reinicializaçao.

Exemplos:

```
for(i=0; i<10; i++) {
     a -= i
}

for(i = 0; j = 10; !j; ) {
     i++;
}

for(a = 1; a != b; a+=5) {
     b += c * a;
```

```
        c++;
    }

    for(; ; ); /* comando vazio – laço eterno */
```

Comando `break`

O comando break é usado para interromper o laço dos comandos while, for e do...while.

Comando `continue`

O comando continue provoca uma nova avaliação da expressão de controle dos comandos de repetição.

DIRETIVAS DE COMPILAÇÃO

O Pré-processador

Conceitualmente, o pré-processador C é um filtro que é executado antes do compilador. Ele executa substituições textuais, inclusão de arquivos, e é capaz de processamento condicional. Todas as diretivas para o pré-processador começam como sinal cardinal #, que deve ocorrer no começo da linha.

Diretiva #define

Existem duas formas da diretiva define. A primeira é #define identificador string-chave e faz com que o identificador seja substituído pela cadeia de caracteres chave onde quer que ele ocorra no módulo fonte:

```
#define identificador(identificador,...) string-chave
```

A segunda forma é:

Não é permitido espaço em branco entre o primeiro identificador e o parênteses esquerdo. Esta forma é capas de substituições mais complexas; assim

```
#define min(A,B) A<B?A:B
```

define uma substituição da expressão min(a,b) que resulta no menor dos dois arqumentos. Porque você está lidando com substituição textual, os argumentos da macro pode ser de qualquer tipo válido para as operações. Existem, no entanto, dois problemas. Primeiro, a macro pode ter efeitos colaterais; assim, min(++a,b) expande-se para a++< b ? ++a:b que faz mais do que era esperado. Segundo, já que um identificador pode ser substituído por uma expressão contendo operadores, é aconselhável colocar cada identificador entre parênteses. Logo

```
#define min(A,B) ((A)«B)?(A):(B»
```

é mais seguro.

Diretiva #include

A diretiva include tem a forma:

```
#include "nome-do-arquivo"
```

ou

```
#include <nome-do-arquivo>
```

A linha é substituída por todo o conteúdo do arquivo especificado. As duas formas podem, ou não. ser tratadas de forma identicas. Em ambos os casos, os arquivos referenciados são obtidos em algum lugar pré-determinado no sistema; a primeira forma.

pesquisa primeiros no seu próprio diretório. Como regra geral, use < > para incluir arquivos gerais do C, como:

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

VETORES

Um vetor é uma coleção de variáveis do mesmo tipo que são referenciadas pelo mesmo nome. Na Linguagem C, um vetor consiste em locações contíguas de memória. O endereço mais baixo corresponde ao primeiro elemento, e o mais alto corresponde ao último elemento. Um vetor pode ter de uma a várias dimensões. Você acessa um determinado elemento em um vetor usando índice.

O vetor que você vai usar é a de caracteres. Como C não tem tipo de dados *string* intrínseco, ele usa os vetores de caracteres. Como você verá em breve esse método de lidar com strings concede mais força e flexibilidade que as linguagens que usam tipos strings especiais.

Vetores de Uma Dimensão

A forma geral da declaração de um vetor é:

```
tipo nome-var [tamanho];
```

Aqui, tipo declara o tipo base do vetor. O tipo base determina o tipo de dado de cada elemento do vetor. O tamanho define quantos elementos o vetor conterá. Por exemplo, segue.se a declaração de um vetor de inteiros chamado amostra que tem tamanho de 10 elementos:

```
 int amostra[10];
```

Em C, todas os vetores usam zero como índice do primeiro elemento. Portanto o exemplo acima declara um vetor de inteiros que terá 10 elementos: amostra[0] até amostra[9]. Por exemplo, o programa a seguir carrega um vetor de inteiros com números de 0 a 9:

```
main() {
     int x[10];     /* isto reserva 10 elementos inteiros */
     int t;
     for (t=0; t<10; ++t) {
          x(t]=t;
     }
}
```

Para um vetor de dimensão única, você computa o tamanho de uma matriz em bytes. conforme mostramos aqui:

```
total-de-bytes = sizeof(tipo) * comprimento-do-vetor
```

Os vetores são comuns em programação porque permitem que você lide facilmente com muitas variáveis relacionadas. Por exemplo, o uso de matrizes facilita calcular a média de uma lista de números, conforme mostra este programa:

```
/* encontrar a média de dez números */
```

```
main() {
     int amostra[10], i, med;
     for (i=0; i<1O; i++) {
          printf("digite número %d: ", i);
          scanf ("%d", &amostra[i]);
     }
     med=0;
     /* agora somar os números */
     for (i=O; i<1O; i++) {
          med = méd + amostra[i];
     }
     printf("A média é %d\n", medl1 C);
}
```

Sem verificação de limites

A linguagem C não realiza verificação de limites em vetores; por isso, nada impede que você vá além do fim de um vetor. Se você transpuser o fim de um vetor durante uma operação de atribuição, então você atribui valores a dados de outras variáveis ou até mesmo a uma parte do código do programa. Eis aqui uma outra maneira de visualizar esse problema: você pode indexar um vetor de tamanho N além de N sem nenhuma mensagem de erro em tempo de compilação ou de erro em tempo de execução, embora esse ato possa causar um erro em seu programa. Como programador, você tem a responsabilidade de assegurar que todos os vetores sejam grandes o bastante para conter tudo o que o programa irá colocar dentro deles; você também tem a responsabilidade de providenciar a verificação dos limites sempre que necessário. Por exemplo, Turbo C irá compilar e rodar este programa, embora ele transponha o vetor erro:

```
/* Um programa incorreto /*
main() {
     int erro[10], i;
     for (i=0; i<100; i++) {
          erro[i] = i;
     }
}
```

Nesse caso, o laço (loop) terá 100 iterações, embora o vetor erro tenha apenas 10 elementos de tamanho.

Você pode se perguntar por que Turbo C ou a linguagem C em geral, não fornece a verificação de limites em vetores. A resposta é que C foi projetado para substituir na maioria das situações a codificação em linguagem assembly. Para fazer isto, C não inclui virtualmente nenhuma verificação de erro, porque isto aumenta (geralmente de forma dramática) o tempo de execução do programa. Em vez disso, C espera que o programador seja bastante responsável para evitar em primeiro lugar a transposição de vetores.

Vetores unidimensionais são conjuntos

Os vetores de dimensão única são essencialmente um conjunto de informações do mesmo tipo. Por exemplo, depois que você rodar este programa:

```
main () {
     char ch[7];
     int i;
     for (i=O; i<6; i++) {
          ch[i] = 'A' + i;
     }
     ch[6] = '\O';
}
```

ch ficará assim:

ch[0]	ch[1]	ch[2]	ch[3]	ch[4]	ch[5]	ch[6]
A	B	C	D	E	F	G

Vetores bidimensionais ou matrizes

C permite vetores multidimensionais. A forma mais simples do vetor multidimensional é a bidimensional ou matriz. Um vetor bidimensional é um vetor de vetores unidimensionais. Para declarar uma matriz de inteiros bindm de tamanho 10 x 20 você escreverá:

int b1dm[10][20];

Exemplo:

```
main() {
     int t, I, num[3][4];
     for (t=O; t<3; ++t) {
          for (i=0; i<4; ++i) {
               num[t][i] = (t*4) + i + 1 ;
          }
     }
}
```

Neste exemplo, num[0][0] terá valor 1, num[0][1] terá valor 2, num[0][2] terá valor 3 e assim por diante. O valor de num[2][3] será 12.

STRINGS (Cadeias de Caracteres)

Até agora, o uso mais comum dos vetores unidimensionais é criar strings (cadeias) de caracteres. Em C, um string consiste em um vetor de caracteres terminada em zero. Um zero é especificado como '\0'. Por essa razão, você deve declarar os vetores de caracteres como sendo um caractere maior que o maior string que você quer que eles contenham. Por exemplo, se você declarar um vetor str que conterá um slring de dez caracteres, você escreverá:

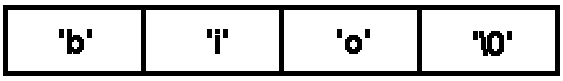
```
char str[11];
```

Esta declaração deixa espaço para o zero no final do string.

O C, embora não tenha o tipo de dado string, ainda assim permite constantes string. Lembre-se que uma constante string é uma lista de caracteres que aparecem entre aspas. Eis aqui dois exemplos:

```
"bom dia"
"isto é um teste"
```

Você não precisa acrescentar o zero manualmente no final das constantes string. O compilador C faz automaticamente. Assim, o string "bio" ficará na memória da seguinte maneira:



Lendo um string pelo teclado

A melhor maneira de inserir uma string através do teclado é usar a função de biblioteca gets(). A forma geral de gets() é:

```
gets(nome-do-vetor);
```

Para ler um string, você chama gets() com o nome-do-vetor, sem qualquer índice como argumento. De acordo com o retorno de gets(), o vetor conterá o string introduzido pelo teclado. A função gets() continuará até que você insira um carriage return.

```
void main() {
     char str[80];

     gets(str);
     printf("%s", str);
}
```

Note que você pode usar str como argumento de printf(). Note também que o programa usa o nome do vetor sem um índice. Por motivos que ficarão claros

posteriormente, você também pode usar o nome de um vetor de caracteres sem índice, desde que ela contenha um string em qualquer lugar onde você possa usar uma constante string.

Tenha em mente que gets() não efetua nenhuma verificação de limites no vetor com o qual ela e chamada. Portanto, se você inserir um string maior que o tamanho do vetor, esse vetor será sobrescrito.

Algumas funções de bibliotecas para usar com strings

A função strcpy()

A chamada da função strcpy() tem a forma geral:

```
strcpy(destino, origem);
```

Você usa a função strcpy() para copiar o conteúdo do string origem para destino.

Lembre-se de que o vetor destino deve ser grande o bastante para conter o string em origem. Se destino não for suficientemente grande, o vetor será transposto e provavelmente danificará seu programa. Por exemplo:

```
void main() {
     char str[80];

     strcpy(str, "bio");
}
```

A função strcat()

Para se chamar a função strcat(), usa-se esta forma:

strcat(s1, s2)

A função strcat() anexa s2 ao final de s1; s2 fica inalterada. Ambos os strings devern ser encerrados por um ‘\0’, e o resultado também será encerrado por ‘\0’.

```
void main() {
     char primeiro[20], segundo[10];

     strcpy(primeiro, "bom");
     strcpy(segundo. "dia");
     strcpy(primeiro, segundo);
     printf("%s", primeiro);
}
```

A função strcmp()

Para chamar a função strcmp(), usa-se esta forma geral:

strcmp(s1, s2);

A função strcmp() compara dois strings e devolve 0 se eles forem iguais. Se s1 for lexicograficamente maior que s2, então a função devolverá um número positivo; se s1 for menor que s2, a função devolverá um número negativo.

A função strlen()

A forma geral da chamada de uma função strlen() é strlen(s) onde s é um string. A função strlen() devolve o comprimento de um string para a qual s aponta.

Exemplo:

```
void main() {
     char str[80];

     printf(“digite um string: “)
     gets(str);
     printf("%d", strlen(str));
}
```

Usando o terminador ‘\0’ (NULL)

Você pode fazer bom uso do fato de todos os strings terminarem em ‘\0’ para simplificar várias operações com strings. Por exemplo, veja como você precisará de pouco código para tornar maiúsculo cada caractere de uma string:

```
void main() {
     char str[80];
     int i;

     strcpy(str, "isto é um teste");
     for (i=0; str(i); i++) {
          str[i] = toupper(str[i];
     }
     printf("%s", str);
}
```

APONTADORES

Um apontador é uma variável que contém um endereço de memória. Ou seja, esse endereço é a localização de uma outra variável na memória. Se uma variável contém o endereço de uma outra variável, então detemos que a primeira aponta para a segunda. Analise o esquema abaixo:

Endereço na memória	Variável na memória
1000	1003
1001	
1002	
1003	
1004	

Memória

Variável Apontador

Se uma variável irá conter um apontador, então você precisa declara-la como uma variável do tipo apontador. A forma geral é:

```
tipo *nome-da-variável
```

onde o tipo pode ser qualquer tipo base válido em C, e nome-da-variável é um identlficador válido. Por exemplo, estes comandos declaram apontadores a inteiros e caracteres:

```
char *p;
int *temp, *inicio;
```

Operadores de Apontadores

Existem dois operadores especiais de apontadores: ***** e **&**, que são operadores unários. Por exemplo:

```
x = &y;
```

Coloca em x o endereço de memória da variável y. Esse endereço é a localização interna da variável na memória do computador. O endereço não tem nada a ver com o valor de y. Você pode se lembrar da operação de & devolvendo "o endereço de" para a

variável que precede. Portando você pode ler o comando de atribuição que acabamos de mostrar como "x recebe o endereço de y".

Para entender essa atribuição mais claramente, assuma que a variável y esteja localizada no endereço 2000. Depois da atribuição que acabamos de mostrar, x conterá o valor 2000.

O segundo operador, é o complemento de &. Ele é um operador unário que devolve o valor da variável localizada no endereço que se segue. Por exemplo, se x contiver o endereço de memória da variável y, então:

```
k = *x;
```

Coloca o valor de y em k. Por exemplo, se y originalmente contivesse o valor 100, então, depois dessa atribuição, k conteria o valor 100, porque este é o valor armazenado na localização 2000, que é o endereço de memória que foi atribuído a x. Lembre-se da operação de * como "no endereço". Assim. nesse caso, você poderia ler o comando acima como "k recebe o valor que está no endereço x".

O sinal de multiplicação e o sinal "no endereço" são iguais. Ao escrever seus programas, tenha em mente que esses operadores não têm relação uns com os outros. Tanto o operador & como * têm precedência mais alta que todos os outros operadores aritméticos, exceto o menos unário, com o qual eles se equivalem.

Eis aqui um programa que usa os dois comando de atribuição que acabamos de mostrar:

```
void main() {
     int x, *y, k
     x = 100;
     y = &x; /* obtém o endereço de x */
     k = *y; /* obtém o valor naquele endereço */
     printf("%d", k); /* exibe 100 */
}
```

Expressões com Apontadores

Em geral, as expressões que envolvem apontadores obedecem às mesmas regras das outras expressões em C.

Atribuição com Apontadores

Como acontece com qualquer variável, você pode usar um apontador no lado direito dos comando de atribuição para atribuir seu valor a um outro apontador, como neste exemplo:

```
void main() {
     int x = 10;
     int *p1, *p2;
     p1 = &x;
     p2 = p1;
```

```
     printf("%p", p2); /* exibe o valor do endereço de x,
                          não o valor de x */
}
```

Aritmética com apontadores

Em C, você pode ter apenas duas operações aritméticas com apontadores: + e -. para entender o que ocorre na aritmética com apontadores, suponha que p1 seja um apontador para um inteiro com valor corrente de 2000. Depois da expressão:

```
p1++;
```

o conteúdo de p1 será 2002 e não 2001. Cada vez que o computador incrementa p1, ele aponta para o próximo inteiro. O mesmo é verdade para os decrementos. Por exemplo:

```
p1--;
```

fará com que p1 tenha o valor 1998, se anteriormente tivesse o valor 2000.

Cada vez que o computador incrementa um apontador, ele aponta para a localização de memória do próximo elemento de seu tipo-base. Cada vez que o computador decrementa, ele aponta para a localização do elemento anterior. No caso dos apontadores para caracteres, a aritmética com apontadores, em geral, se parece com a "normal". Porém, todos os outros apontadores aumentam ou diminuem de acordo como o comprimento do tipo de dado para o qual aponta.

Inicialização de apontadores

Depois que você declara um apontador, mas antes de lhe atribuir um valor, ele conterá um valor desconhecido. Se tentar usar um apontador antes de atribuir-lhe um valor, você provavelmente danificará não apenas o programa mas também o sistema operacional. Por convenção, você deve dar a um apontador que não aponta para lugar nenhum o valor nulo (null), para indicar esse fato. Porém o fato de um apontador ter simplesmente o valor nulo não o torna "seguro" para ser usado. Se você usar um apontador nulo ao lado esquerdo de um comando de atribuição, ainda correrá o risco de danificar seu programa.

Por enquanto, a forma mais segura de ínicializar um apontador é atribuir-lhe o endereço de uma variável declarada. Como por exemplo:

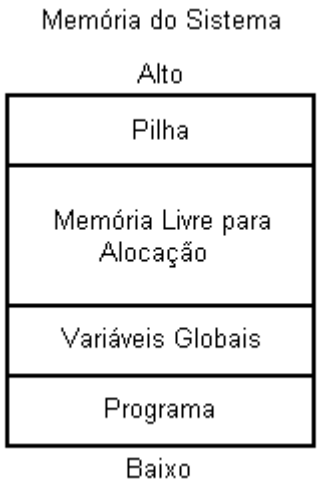
```
int x = 4;
int *y;
y = &x;
```

Alocação dinâmica

Existem dois métodos principais por meios dos quais um programa em C pode armazenar informações na memória principal do computador. O primeiro usa as variáveis

locais e globais que C define. No caso das variáveis globais, o armazenamento é fixado durante o tempo de execução de seu programa. Para essas variáveis, o programa aloca armazenamento no espaço da pilha do computador. Embora as variáveis locais sejam eficientemente implementadas em C. elas exigem que você saiba, de antemão, o montante de memória necessária para cada situação.

O segundo método que o programa pode usar para armazenar informações é através das funções de alocação dinâmica do C, `malloc()` e `free()`. Nesse método um programa aloca armazenamento para informações da área de memória livre chamada heap (ou área de alocação dinâmica), que fica entre seu programa, com sua área de armazenamento permanente, e a pilha. Examine a figura a seguir:



A figura mostra a forma conceitual como um programa aparece na memória. A pilha cresce de cima para baixo na medida em que o programa a usa; portanto o desenho do programa determina a quantidade de memória que ele precisa. Por exemplo, um programa com muitas funções recursivas demandará muito mais memória da pilha que um programa que não tenha funções recursivas, porque as variáveis locais e os endereços de retorno estão armazenados na pilha. Lembre-se de que o computador aloca permanentemente a memória exigida para o programa e dados globais durante a sua execução. O computador obtêm memória da área de memória livre para satisfazer um pedido de `malloc()`, iniciando exatamente acima das variáveis globais e crescendo em direção à pilha.

As funções malloc() e free()

As funções `malloc()` e `free()` formam o sistema de alocação dinâmica do C e são parte de sua biblioteca. Elas trabalham juntas usando a região de memória livre que fica entre seu programa e o começo da pilha, para estabelecer e manter uma lista de memória disponível.

Cada vez que você fizer um pedido de memória `malloc()`, esta alocará uma parte do restante da memória livre. Cada vez que você pedir uma liberação de memória `free()`, esta devolverá memória para o sistema.

A função `malloc()` retorna um apontador do tipo `void`, que significa que você deverá usar um *cast* explícito quando atribuir o apontador devolvido a um apontador do tipo que você deseja. Depois de uma chamada bem sucedida, `malloc()` devolverá um apontador para o primeiro byte da região de memória que foi alocada. Se não houver memória suficiente disponível para satisfazer o pedido `malloc()`, ocorrerá um fracasso na alocação e `malloc()` devolverá um apontador nulo. Você pode usar `sizeof` para determinar O número exato de bytes de que cada tipo de dado precisa. Fazendo isto, você torna seus programas portáteis a uma variedade de sistemas.

Depois de usar free() para liberar memória, você pode reusá-la através de uma seqüente chamada a malloc1(). Exemplo:

```
void main() {
     int *p, t;
     p = (int *) malloc(40 * sizeof(int));
     if (!p) {
     printf(“memória insuficiente\n”);
     }
     else {
     for (t = 0; t < 40; ++t) {
     *(p+t) = t;
     }
     for (t = 0; t < 40; ++t) {
     prlntf("%d ", *(p+t));
     }
     free(p);
     }
}
```

FUNÇÕES

Sintaxe:

```
tipo-r nome-da-função(lista-de-declarações-dos-parâmetros) {
     declaração-de-variáveis-locais;
     corpo-da-função;
}
```

O tipo-r especifica tipo da expressão de retorno. Por *default*, quando o tipo-r é omitido, o C assume que a função é do tipo `int`, ou seja a expressão de retorno é inteira. Quando uma função não retorna nenhum valor, dizemos que ela é do tipo `void`. Exemplo:

```
max(int x, y) {
     int aux;
     if (x > y) {
          aux = x;
     }
     else {
          aux = y ;
     }
     return(aux);
}
```

A função acima retornará o maior valor entre dois números inteiros. Uma possível chamada dessa funçâo seria:

```
          printf("O maior valor entre %d e %d é %d",
                 a, b, max(a,b));
```

Todos sabemos que há duas formas de chamar uma função. A primeira forma é chamada por valor (ou passagem por valor). Esse método copia o valor dos argumentos nos parâmetros formais (parâmetros declarados na função). A função max() declarada acima é um exemplo de chamada por valor.

A segunda forma de passagem de argumentos para uma função é chamada por referência, onde o endereço de cada argumento é compartilhado pelos parâmetros formais. Isso significa que as alterações feitas nos parâmetros afetam a os respectivos argumentos (variáveis) utilizados para chamar a função.

Na chamada por referência os parâmetros devem ser declarados como apontadores Veja o exemplo abaixo:

```
troca(int x, y) {
     int aux;
     aux = *x;
     *x = *y;
     *y = aux;
}
```

ESTRUTURAS

Estruturas são grupos de variáveis. Diferentemente de matrizes, podem ser compostas de tipos diferentes (inteiros, reais, ...). Elas são comparáveis aos registro em outras linguagens. Cada estrutura pode ser tratada como um todo ou por cada um de seus componentes.

Sintaxe:

```
struct rótulo {
     declaração-de-variável;
     nome-da-variável;
};
```

O rótulo é opcional e será discutido posteriormente. As declarações de variáveis são as mesmas já vistas. O nome da estrutura segue as mesmas regras dos nomes de variáveis simples. Por exemplo:

```
struct {
     char mes[10];
     int dia;
     int ano;
} feriado;
```

esse exemplo declara feriado como sendo uma estrutura que tem três variáveis.

Operador de Seleção

Cada membro da estrutura é o equivalente a uma variável simples de seu tipo. o operador de seleção (.) é utilizado para referenciar um membro de um estrutura. Por exemplo:

```
feriado.mes[0] = 'B';
feriado.dia = 1;
feriado.ano = 94;
```

Rótulo

Uma estrutura pode receber um rótulo. Uma vez definida, ela pode ser utilizada para declarar outras estruturas como os mesmos membros. Veja o seguinte exemplo:

```
struct data {
     char mes[10];
     int dia;
     int ano;
};
```

Uma vez que data esteja definida, ela pode ser utilizada para declarar outras variáveis. Por exemplo:

```
struct data diaDoAno, feriado;
```

Inicialização de Estruturas

A iniclalização de estruturas assemelha-se muito como a inicialização de matrizes:

```
struct data feriadao = {"FEVEREIRO", 1, 1994};
```

ARQUIVOS

Os arquivos em C precisam ser explicitamente abertos e referenciados. As funções fopen() e fclose() são utilizadas, respectivamente para abrir e fechar arquivos.

Cada arquivo aberto é referenciado por um identificador de arquivo que deve ser do tipo FILE (estrutura definida no arquivo stdio.h no diretório INCLUDE).

Exemplo de declaração de um identificador de arquivo:

```
FILE *id-arq;
```

A tabela abaixo mostra algumas das funções e macros para manipulação com arquivos:

FUNÇÃO	*AÇÃO*
fopen	abre um arquivo
fwrite	escreve dados em um arquivo
Fread	ler dados de um arquivo
fclose	fechar um arquivo
Feof	verifica se o apontador do arquivo atingiu a marca final do arquivo

Função fopen()

Sintaxe:

```
fopen("nome-arquivo", “modo-de-acesso");
```

Abre um arquivo e retorna um estrutura FILE que identificará arquivo ou stream associado. Se fopen() não abrir o arquivo, um NULL é retornado.

O modo de acesso é especificado pelos caracteres:

“r"	abre um arquivo para leitura e o apontador é posicionado no início-do-arquivo.
"w”	abre um arquivo apenas para gravação; o arquivo será criado se não existir; se o arquivo existir, todo o conteúdo do arquivo será perdido (reset).
“a"	abre um arquivo no modo de inclusão; se o arquivo não existir, um arquivo será criado; caso exista, todas inclusões serão feitas no final do arquivo.

É ilegal ler de arquivos que estejam abertos nos modos "w" ou "a", ou gravar em arquivos abertos no modo "r". Caso se deseje abrir um arquivo tanto para leitura como para escrita, deve-se usar um "+" logo após o modo desejado. Assim temos os seguintes modos:

"r+"	abre um arquivo para leitura; o apontador é posicionado no início-do-arquivo; gravação também é permitida

	arquivo; gravação também é permitida.
"w+”	abre um arquivo para gravação; leitura também é permitida.
"a+"	abre um arquivo no modo de inclusão; leitura também é permitida.

Função fwrite()

Sintaxe:

```
fwrite(p, bytes, número, stream);
```

A função fwrite() é usada para gravar um número de itens, do tamanho especificado em bytes, no stream (não traduzido), começando na localização de memória p. A função retorna um número inteiro informando a quantidade de bytes realmente gravados. Se um zero for retornado, houve um erro na operação.

Função fread()

Sintaxe:

```
fread(p, bytes, número, stream);
```

Para ler os dados gravados com fwrite() deve-se usar a função fread(). A função retornará um zero se ocorrer um erro, Caso contrário, o numero de bytes realmente gravados.

Macro feof()

Retorna um inteiro deferente de zero ser o fim-de-arquivo foi lido no stream, ou seja, se o apontador de arquivo está posicionado no EOF (End-Of-File).

Função fclose()

Sintaxe:

```
fclose(stream)
```

Fechar um arquivo descarregando o buffer a ele associado.

Exemplo

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

char buffer[] = "Núcleo de Informátlca”

int main (void) {
     FILE *id-arq;
     char buf-str[22];
     /* abre o arquivo no modo de escrita */
```

```
     if ((id-arq = fopen("TEXTO.DAT, “w” )) == NULL) {
          printf("Erro na abertura do arquivo TEXTO.DAT");
          return 1;
     }
     /* escreve um string no arquivo */
     if (fwrite(buffer, sizeof(buffer), 1, id-arq) != 1) {
          printf( "Erro de escrita no arquivo TEXTO.DAT")
          return 1;
     }
     /* fecha o arquivo */
     if (fclose(id-arq) != 0) {
          printf("Erro no fechamento do arquivo TEXTO.DAT");
          return 1;
     }
     /* abre o arquivo no modo de inclusão e leitura */
     if ((id-arq = fopen("TEXTO.DAT", "a+")) == NULL) {
          printf("Erro na abertura do arquivo TEXTO.DAT")
          return 1;
     /* lê um string do arquivo */
     if (fread(buf_str, 22, 1, id-arq) != 1) {
          printf("Erro de leitura no arquivo TEXTO.DAT¨);
          return 1;
     }
     /* adiciona final de string */
     buf-str[22] = '\0';
     printf("\n%s", buf-str);
}
```