SENAC – Serviço Nacional de Aprendizagem Comercial
Curso de Programação em C

Linguagem C

Florianópolis
2003

Linguagem C

Marcelo Buscioli Tenório

Curso de Programação em C
SENAC

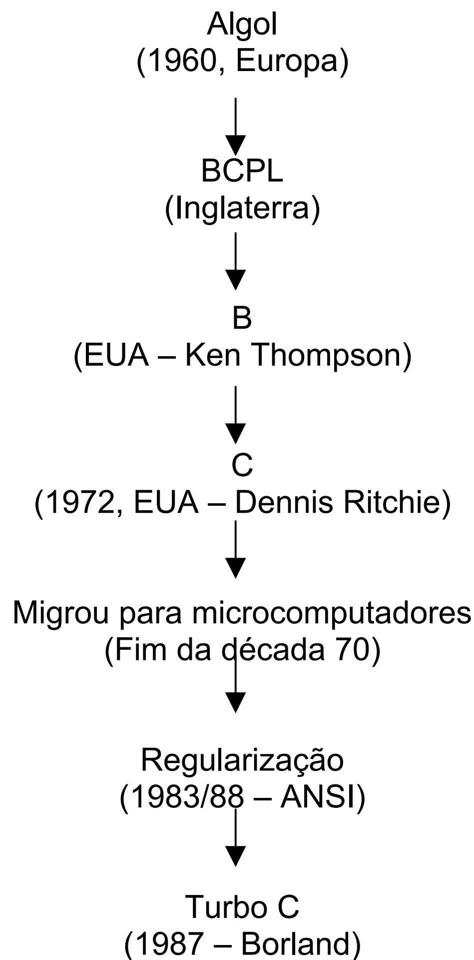
1 PREFÁCIO 5
Histórico 5
Características 6
2 ESTRUTURA BÁSICA DE UM PROGRAMA EM C 7
Regras Gerais 7
Forma Geral de Funções 7
3 VARIÁVEIS 8
Regras Gerais 8
Palavras Reservadas 9
Tipos de Variáveis 9
Declarando Variáveis 9
Inicializando Variáveis 9
Regras Gerais 10
4 OPERADORES 10
Aritméticos 10
Relacionais 10
Lógicos 10
Bit a bit 11
5 ENTRADA/SAIDA PELO CONSOLE 11
Funções de saída 11
Funções de entrada 13
6 ESTRUTURAS DE DECISÃO 13
if 13
?: 14
switch 14
7 ESTRUTURAS DE REPETIÇÃO 15
for 15
while 16
do... while 17
Comandos Auxiliares 17
8 FUNÇÕES 18
Declaração 18
Tipos 18
9 CLASSES DE ARMAZENAMENTO 19
auto 19
extern 20
static 20
register 20
10 DIRETIVAS 21
#define 21
#undef 21
#include 21

#if, #ifdef, #ifndef, #else, #endif 22
11 VETORES E MATRIZES 22
Vetor / Matriz 22
12 STRINGS 23
Lendo Strings 24
Imprimindo Strings 24
Funções de Manipulação de Strings 24
13 ESTRUTURAS 25
Acessando a Estrutura 26
Matriz de Estrutura 26
Inicializando uma Estrutura 26
14 PONTEIROS 26
Ponteiro como Parâmetro de Função 28
Matriz de Ponteiro 28
Ponteiro para Estrutura 29
Alocação Dinâmica 29
15 ENTRADA/SAIDA PELO ARQUIVO 31
Streams e Arquivos 31
As Funções mais Comuns 31
Usando fopen(), fgetc(), fputc() e fclose() 32
Trabalhando com Strings: fputs() e fgets() 32
Usando fread() e fwrite() 33
BIBLIOGRAFIA 35

1 PREFÁCIO

Esta apostila foi elaborada como guia de ensino da linguagem C, por isso mostra-se de modo superficial e pouco detalhada.

Histórico



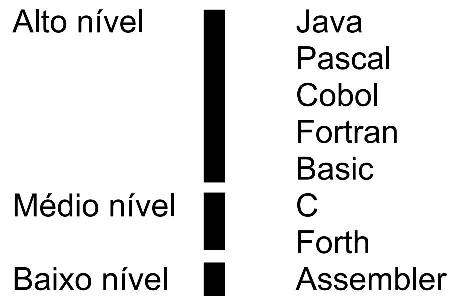
A linguagem C foi primeiramente criada por Dennis M. Ritchie e Ken Thompson no laboratório Bell em 1972, baseada na linguagem B de Thompson que era uma evolução da antiga linguagem BCPL.

A linguagem ficou contida nos laboratórios até o final da década de 70, momento que começou a popularização do sistema operacional UNIX e consequentemente o C (o UNIX é desenvolvido em C).

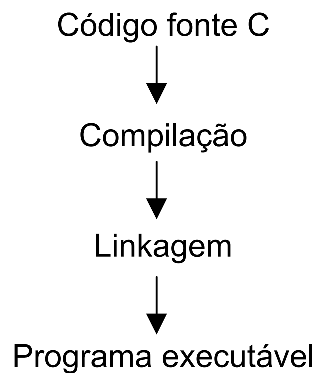
Os programas escritos em Assembler, como o dBase II e III, Wordstar, Lotus 1.0 dente outros, tiveram seus fontes convertidos em C, originando como exemplo o dBase III Plus, Lotus 2.0, MS-Word e MS-Excel. É utilizado amplamente no desenvolvimento de Sistemas Operacionais, Sistema Gerenciador de Banco de Dados, aplicativos e utilitários.

Características

- O C une conceitos de linguagem de montagem e programação de alto nível (o programador usufrui recursos de hardware sem a necessidade de conhecer o assembly da máquina);



- Linguagem compilada;



- Linguagem estruturada;

Estruturada	Não estruturada
Pascal	Fortran
ADA	Basic
C	Cobol

- Possui poucos comandos, cerca de 32 na versão básica e 43 na versão para o Turbo C;
- Não possui crítica eficiente para erros de execução;
- Uma linguagem em constante evolução. Existem versões orientadas a objeto e visuais (C++ e C++ Builder).

"C é uma linguagem para programadores"
Herbert Schildt

2 ESTRUTURA BÁSICA DE UM PROGRAMA EM C

Um programa C consiste em uma ou várias funções, uma das quais precisa ser denominada *main*. O programa sempre começará executando a função *main*. Definições de funções adicionais podem preceder ou suceder a função *main*.

Regras Gerais

- Toda função deve ser iniciada por uma chave de abertura ({) e encerrada por uma chave de fechamento (});
- Toda função é procedida de parênteses "( )";
- Todo programa deverá conter a função *main*;
- As linhas de código são sempre encerradas por um ponto e vírgula;
- A formatação dos programas é completamente livre, mas temos por conveniência manter a legibilidade;
- Os comandos são executados na ordem em que foram escritos;
- Os comentários devem ser delimitados por /* no início e */ no final. Podem ser usados também os caracteres // para comentários de uma linha.

Forma Geral de Funções

```
/* Os comentários podem ser colocados em qualquer parte do programa */

declaração de variáveis globais

void main(void)
{
  declaração de variáveis locais
  ---
  --- comandos;
  ---
}

tipo função(lista dos argumentos)
{
  declaração das variáveis locais a função
  ---
  --- comandos;
  ---
}
```

Exemplo

```
/* Calcula a área de um circulo */

#include <stdio.h>
#include <conio.h>

float processa(float r);                // declaração da função processa

void main(void)                         // função principal main
{
  float raio, area;                     // declaração de variáveis locais
  clrscr();
  printf("Raio: "); scanf("%f", &raio);
  area = processa(raio);
  printf("Área: %f", area);
  getche();
}

float processa(float r)                 // implementação da função processa
{
  float a;                              // declaração de variáveis locais
  a=3.1415*r*r;
  return(a);
}
```

3 VARIÁVEIS

É um espaço de memória que pode conter, a cada tempo um valor diferente.

Regras Gerais

- Em C, todas variáveis utilizadas no programa devem ser declaradas previamente;
- A declaração indica, no mínimo, o nome e o tipo de cada uma delas;
- Na ocorrência de mais de uma variável do mesmo tipo, podemos declara-las de uma única vez, separando seus nomes por vírgula;
- As letras minúsculas são diferenciadas das maiúsculas;
- O tamanho máximo significante para uma variável é de 31 posições;
- Todo nome de variável é iniciado por uma letra (a – z ou A – Z), ou o caracter traço baixo (_), o restante pode conter letras, traço baixo ou números;
- As palavras reservadas descritas a seguir, não podem ser usadas como nome de variável.

Palavras Reservadas

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

Exemplos de nome de variáveis: a, b, VALOR1, valor_1, _nome, ano_2003

Tipos de Variáveis

Tipo	Bits	Escala
void	0	sem valor
char	8	-128 a 127
int	16	-32768 a 32767
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
long int	32	3.4E-38 a 3.4E+38
unsigned char	8	0 a 255
unsigned int	16	0 a 65535

Declarando Variáveis

A forma de declaração de variáveis é a seguinte: ***tipo* nome;**
Exemplos

```
int K;
double valores;
float quadro;
char caracter;
```

Inicializando Variáveis

Pode-se inicializar uma variável no momento de sua declaração, ou em qualquer momento do algoritmo.

```
int c=8;                 // global
void main(void)
{
  float tempo=27.25;     // local
  char tipo='C';         // local
  int x;
  x=10;
  c=c+x;
}
```

Regras Gerais

- As variáveis deverão ser declaradas antes de qualquer instrução;
- As variáveis do tipo caracter deverão receber valores entre aspas simples (‘);
- Não existe o tipo booleano (falso, verdadeiro) nem o tipo *string*, o valor zero (0) representa um valor falso, os demais valores (1, ’a’, ‘z’, -1, -0.7) são considerados verdadeiros.

4 OPERADORES

Aritméticos

Sinal	Significado	Exemplos
=	atribuição	a=5; c=’B’;
+	soma	a=a+1; tot=salario+comissao;
-	subtração	tot=subtotal-desconto;
*	multiplicação	nota=nota*2;
/	divisão	comissao=lucro/4;
%	módulo (resto da divisão)	resto=8%3;
++	incrementa de 1	i++;
--	decrementa de 1	j--;

Relacionais

Sinal	Significado
>	maior
>=	maior ou igual
<	menor
<=	menor ou igual
==	igual
!=	diferente (não igual)

Lógicos

Sinal	Significado
&&	e (and)
||	ou (or)
!	não (not)

Bit a bit

<table>
<tr><th>Sinal</th><th>Significado</th><th>Exemplo</th></tr>
<tr><td>~</td><td>negação</td><td>a=0000 0000 0110 1001 ➔ 105
~a=1111 1111 1001 0110 ➔ -106</td></tr>
<tr><td>&</td><td>and</td><td>a=0000 0000 0110 1001 ➔ 105
b=0000 0010 0101 1100 ➔ 604
a&b=0000 0000 0100 1000 ➔ 72</td></tr>
<tr><td>|</td><td>or</td><td>a=0000 0000 0110 1001 ➔ 105
b=0000 0010 0101 1100 ➔ 604
a | b=0000 0010 0111 1101 ➔ 637</td></tr>
<tr><td>^</td><td>or exclusivo</td><td>a=0000 0000 0110 1001 ➔ 105
b=0000 0010 0101 1100 ➔ 604
a^b =0000 0010 0011 0101 ➔ 565</td></tr>
<tr><td><<</td><td>deslocamento para esquerda</td><td>a=0000 0010 0101 1100 ➔ 604
a<<3=0001 0010 1110 0000 ➔ 4832</td></tr>
<tr><td>>></td><td>deslocamento para direita</td><td>a=0000 0010 0101 1100 ➔ 604
a>>3=0000 0000 0100 1011 ➔ 75</td></tr>
</table>

5 ENTRADA/SAIDA PELO CONSOLE

Funções de saída

printf("expressão de controle", lista de argumentos);

A função *printf()* pode ter um ou mais argumentos, a expressão de controle pode conter caracteres que serão exibidos na tela e códigos de formatação que indicam o formato que os argumentos devem ser impressos.

Códigos de barra invertida

Código	Significado
\n	nova linha (LF)
\a	alerta (beep)
\f	nova tela ou nova página (FF)
\t	tab
\b	retrocesso (BS)
\0	nulo
\"	aspas duplas
\'	aspas simples
\\	barra invertida

Códigos de formatação

Código	Significado
%c	caracter
%d	inteiro decimal com sinal
%i	inteiro decimal com sinal
%e	notação científica
%f	ponto flutuante
%o	octal
%s	string de caracteres
%u	inteiro decimal sem sinal
%x	hexadecimal

Exemplo

```
void main(void)
{
  char cidade_A='X', cidade_B='Y';
  float distanc=25.5;
  printf("Cidade = %c \n", cidade_A);
  printf("Distancia = %f Km\n", distanc);
  printf("A %s %c esta a %f Km da %s %c",
        "cidade", cidade_A, distanc, "cidade", cidade_B);
}
```

Tem-se

```
       Cidade = X
       Distancia = 25.5 Km
       A cidade X esta a 25.5 Km da cidade Y
```

putchar(variável caracter);

A função *putchar()* exibe um caracter.

Exemplo

```
void main(void)
{
  char letra='a';
  putchar(letra);
}
```

Funções de entrada

scanf("expressão de controle", lista de argumentos);

A expressão de controle utiliza os mesmos códigos de formatação da função *printf()* descritos acima.

A lista de argumentos deve consistir dos endereços das variáveis. A linguagem oferece um operador para tipos básicos chamado operador de endereço e é referenciado pelo símbolo & que retorna o endereço da variável.

Exemplo

scanf("%f", &x); → lê a variável x do tipo float

getche()

Lê um caracter do teclado e exibe na tela. Não requer o pressionamento da tecla Enter após o caracter digitado.

Exemplo

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

getch()

Semelhante a função *getche()*, mas não exibe o caracter lido na tela.

Exemplo

```
ch=getch();
```

6 ESTRUTURAS DE DECISÃO

if

O comando *if* instrui o computador a tomar uma decisão simples. Se o valor entre parênteses da expressão de controle for verdadeiro ele executa as instruções, caso seja falso, as instruções serão ignoradas, ou executadas as que estão contidas no bloco do *else*.

```
if (expressão de teste)
  instrução1;
else
  instrução2;
```

Exemplos

1)
```
if (X%2)
  printf(" X é impar ");
else
  printf(" X é par ");
```

2)
```
if (X<Y)
  if (X<Z)
    printf(" X é o menor ! ");
  else
    printf(" Z é o menor ! ");
else
  if (Y<Z)
    printf(" Y é o menor ! ");
  else
    printf(" Z é o menor ! ");
```

?:

O operador ?: (condição ternária) é, como o nome indica, um operador que necessita de três operandos, que podem ser resultados de expressões. Em certas situações pode substituir a instrução *if*.

expressão_1 ? expressão_2 : expressão_3

Tem como resultado o valor de expressão_2 ou de expressão_3 consoante o valor de expressão_1 for verdadeiro (!=0) ou falso (==0) respectivamente.

Exemplo

```
z = (a > b) ? a : b;
```

switch

A instrução *switch* faz com que um determinado grupo de instruções seja escolhido entre diversos grupos disponíveis. A seleção é baseada no valor corrente de uma expressão incluída na instrução *switch*.

O corpo de cada *case* é composto por qualquer número de instruções. Geralmente a última instrução é o *break*. O comando *break* causa a saída imediata de todo o corpo do *switch*. Na falta do comando *break*, todas as instruções após o *case* escolhido serão executadas.

Quando a instrução *switch* é executada, a expressão é avaliada e o controle é transferido diretamente para o grupo de instruções correspondente ao valor de expressão. Se nenhum *case* corresponder ao valor de expressão, não será selecionado nenhum dos grupos da instrução *switch*, exceto quando for implementado o grupo *default*, que contém instruções que serão executadas caso não seja selecionado nenhum dos grupos.

```
switch (variável)
{
  case valor1:
      instruções;
      break;
  case valor2:
      instruções;
      break;
  case valor3:
      :
  default:
      instruções;
}
```

Exemplo

```
switch (opcao = getchar())
{
  case ‘V’:
    printf(“ Vermelho “);
    break;
  case ‘B’:
    printf(“ Branco “);
    break;
  case ‘A’:
    printf(“ Azul “);
    break;
  default:
    printf(“ Cor desconhecida ! “);
}
```

7 ESTRUTURAS DE REPETIÇÃO

for

O laço *for* possui três expressões, e é útil principalmente quando queremos repetir algo uma quantidade fixa de vezes.

Dentro dos parênteses após a palavra reservada *for*, há três elementos que controlam a ação da repetição. Em primeiro lugar há uma inicialização que é executada uma única vez. O teste é uma expressão relacional que a função testa no início da repetição. O incremento, é executado no final do *loop* após a execução das instruções.

```
for(inicialização; teste; incremento)
{
  instruções
}
```

Exemplo

```
for(a=1; a<100; a++)
{
  printf("\n %i", a);
}
```

1. O laço *for* inicializa a variável "a" com o valor 1 (a=1)
2. Testa se "a" é menor que 100 (a<100)
3. Se o teste é verdadeiro ele executa as instruções dentro das chaves
4. Incrementa 1 em "a" (a++)
5. Volta a testar a expressão e continua a repetição até o teste ser falso

Outros exemplos

```
for(; ;);          → repetição infinita

for(i=0, j=9; i<=9, j>=0; i++, j--)
  printf("\n %i %i", i, j);
```

while

O laço *while* utiliza os mesmos elementos do laço *for*, mas são distribuídos de maneira diferente no programa.

Se o teste for verdadeiro (diferente de zero), o corpo do laço *while* é executado uma vez e a expressão de teste é avaliada novamente. Este ciclo de teste e execução é repetido até que a expressão de teste se torne falsa (igual a zero), então o laço termina e o controle do programa passa para a linha seguinte ao laço.

```
while (expressão teste)
{
  instruções
}
```

Exemplo

```
while (conta < 10)
{
  total=total + conta;
  printf("conta = %i, total = %i", conta, total);
  conta++;
}
```

do... while

Cria um ciclo repetido até que a expressão de teste seja falsa (zero). A diferença do *while* é que o mesmo testa a expressão, se satisfaz então executa as instruções, enquanto que o *do...while* executa as instruções e depois testa a expressão.

```
do
{
  instruções
}
while (expressão de teste);
```

Exemplo

```
do
{
  y--;
  x++;
}
while (y);
```

Comandos Auxiliares

break

Pode ser usado no corpo de qualquer estrutura de laço. Causa a saída imediata do laço.

Exemplo

```
while (salário>100)
{
  scanf("%s", &nome);
  if (nome == 'a')
    break;
  scanf("%i", &salario);
}
```

continue

O comando *continue* causa um desvio imediato no laço, ou seja, força a próxima interação do laço e ignora o código que estiver abaixo.

Exemplo
```
while (salario>100)
{
  scanf("%c", &nome);
  if (nome == 'a')
  {
    salario=1000;
    continue;
  }
  scanf("%i", &salario);
}
```

8 FUNÇÕES

Uma função é uma unidade de código de programa autônoma desenhada para cumprir uma tarefa particular.

A função recebe um certo número de parâmetros e retorna apenas um valor.

Da mesma forma que são declaradas as variáveis, deve-se declarar a função. A declaração de uma função é chamada de protótipo e é uma instrução geralmente colocada no início do programa, que estabelece o tipo da função e os argumentos que ela recebe.

Declaração

```
tipo nome_função(declaração dos parâmetros)

void main(void)
{
  a=nome_função(lista dos parâmetros);
}

tipo nome_função(lista dos parâmetros)
{
  declaração das variáveis locais
  comandos;
  return(valor);
}
```

Tipos

O tipo de uma função é determinado pelo tipo de valor que ela retorna pelo comando *return* e não pelo tipo de seus argumentos.

Se uma função for do tipo não inteira ela deve ser declarada. O valor default é inteiro (*int*) caso não for declarada.

Exemplos

float – retorna um valor numérico real
int – retorna um valor inteiro
void – sem retorno

```
/* calcula a área de uma esfera */

#define PI 3.14159
float area(int r);

void main(void)
{
  int raio;
  float area_esfera;
  printf("Digite o raio da esfera: "); scanf("%i", &raio);
  area_esfera=area(raio)
  printf("A área da esfera é: %f", area_esfera);
}

float area(int r)
{
  return(4*PI*r*r);
}
```

9 CLASSES DE ARMAZENAMENTO

Todas variáveis e funções em C tem dois atributos: um tipo e uma classe de armazenamento. Os tipos foram apresentados em capítulos anteriores. As classes de armazenamento são quatro:

- auto (automáticas)
- extern (externas)
- static (estáticas)
- register (em registradores)

auto

As variáveis declaradas dentro de uma função são automáticas por padrão. Variáveis automáticas são as mais comuns dentre as quatro classes, elas são criadas quando a função é chamada e destruídas quando a função termina sua execução.

```
void main(void)
{
  auto int i;
}
```

extern

Todas variáveis declaradas fora de qualquer função têm a classe de armazenamento *extern*. Variáveis com esse atributo são conhecidas por todas as funções declaradas depois delas.

```
void main(void)
{
  extrern int x;
}
```

A palavra *extern* não é usada para criar variáveis da classe *extern* e sim para informar ao compilador que uma variável em questão foi criada em outro programa compilado separadamente e será linkeditado junto a este para formar o programa final.

static

Variáveis *static* de um lado se assemelham às automáticas, pois são conhecidas somente nas funções que as declaram e de outro lado se assemelham às externas, pois mantém seus valores mesmo quando a função termina.

Declarações *static* têm dois usos importantes e distintos. O mais elementar é permitir a variáveis locais reterem seus valores mesmo após o termino da execução do bloco onde foram declaradas.

register

A classe de armazenamento *register* indica que a variável associada deve ser guardada fisicamente numa memória de acesso muito mais rápido chamada registrador. Um registrador da máquina é um espaço de 16 bits onde podemos armazenar um *int* ou um *char*.

Basicamente variáveis *register* são usadas para aumentar a velocidade de processamento.

Exemplo

```
#include <time.h>
void main(void)
{
  int i,j;
  register int m, n;
  long t;

  t=time(0);
  for (j=0; j<5000; j++)
    for (i=0; i<5000; i++);

  printf("\n Tempo dos laços não register: %ld", time(0)-t);
```

```
    t=time(0);
    for (m=0; m<5000; m++)
      for (n=0; n<5000; n++);

    printf("\n Tempo dos laços register: %ld", time(0)-t);
  }
```

10 DIRETIVAS

#define

A diretiva *#define* pode ser usada para definir constantes.

Sintaxe: #define <identificador> <valor>

Exemplo

```
#define PI 3.14159
#define begin {
#define end }
#define PRN(n) printf("%0.2f \n", n)

void main(void)
begin
  float num;
  num+=PI;
  PRN(num);
end
```

Quando o compilador encontra a diretiva *#define*, ele procura a ocorrência do identificador e a substitui pelo valor.

#undef

A diretiva *#undef* remove a mais recente definição criada com *#define*.

Exemplo

```
#undef PI     /* cancela a definição de PI */
```

#include

A diretiva *#include*, como o nome sugere, inclui um programa fonte ou um *header file* no código fonte corrente, durante a compilação o compilador substitui a diretiva *#include* do programa pelo conteúdo do arquivo indicado.

Exemplo

```
#include <stdio.h>
#include "funcoes.c"
#include "c:\fontes\mouse.c"

void main(void)
{
  int a, b;
}
```

Quando o arquivo a ser incluído estiver delimitado por "< >", o caminho de procura será o diretório c:\tc\include ou o diretório configurado para conter os *header file*. Quando delimitado por aspas (""), o caminho de procura será o diretório corrente ou o especificado na declaração da diretiva.

Geralmente, os arquivos de inclusão ou *header file* têm o nome com a extensão ".h" e estão gravados no diretório include.

#if, #ifdef, #ifndef, #else, #endif

Estas diretivas são geralmente usadas em grandes programas. Elas permitem suspender definições anteriores e produzir arquivos que podem ser compilados de mais de um modo.

11 VETORES E MATRIZES

Vetor / Matriz

Um vetor ou matriz é um tipo de dado usado para representar uma certa quantidade de valores homogêneos.

Regras

- A linguagem C não valida limites dos vetores, cabe ao programador verificar o correto dimensionamento;
- O primeiro índice é o zero;
- Vetores possuem uma ou mais dimensões, como convenção, os vetores bidimensionais são chamados de matriz;
- O nome do vetor desacompanhado de colchetes representa o endereço de memória onde o mesmo foi armazenado.

Declaração

```
tipo var[tamanho];
```

Exemplos

```
int meses[12];        /* vetor com 12 posições */
```

0	1	2	3	4	5	6	7	8	9	10	11
lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo

float tabela[2][2]; /* matriz de 2x2 */

lixo	lixo
lixo	lixo

Referência aos elementos

meses[0] = 1; /* atribui o valor 1 à primeira posição */

tabela[1][0] = 12.5; /* atribui o valor 12.5 à posição da 2ª. linha e 1ª. coluna */

Inicialização de vetores e matrizes

Podemos fornecer valores a cada posição da matriz na mesma instrução de sua declaração.

int tab[5] = {10, 20, 30, 40, 1};

int mat[ ][ ] = {{10, 10}, {20, 20}};

12 STRINGS

String é uma das mais úteis e importantes formas de dados em C e é usada para armazenar e manipular textos como palavras, nomes e sentenças.

String é um vetor do tipo *char* terminado pelo caracter NULL (\0), ou seja, *string* é uma série de caracteres armazenados em seqüência, onde cada um ocupa um byte de memória, toda *string* é terminada por um byte de valor zero (\0). Cada caracter é um elemento independente e pode ser acessado através de um índice.

Exemplos

char nome[5];
char nome[5] = “João”;
char nome[5] = {‘J’, ‘o’, ‘a’, ‘o’};

0	1	2	3	4
‘J’	‘o’	‘a’	‘o’	‘\0’

char vetor[ ] = “abc”;
char matriz[10][20];
char nome[2][6] = {{‘V’, ‘i’, ‘l’, ‘m’, ‘a’}, {‘D’, ’i’, ‘r’, ‘c’, ‘e’}};
char nome[2][6] = {“Vilma”, “Dirce”};

‘V’	‘i’	‘l’	‘m’	‘a’	‘\0’
‘D’	‘i’	‘r’	‘c’	‘e’	‘\0’

Lendo Strings

scanf()

A função *scanf()* é bastante limitada para a leitura de strings. A função considera o caracter “espaço” como final de string, por exemplo, a leitura do nome Silvia Maria de Jesus pelo *scanf()* (scanf(“%s”, &nome)) resultará em apenas “Silvia”.

gets()

Função própria para leitura de string, a leitura é delimitada pela tecla Enter.

Exemplo

```
gets(nome);  /* nome é uma variável do tipo string */
```

Imprimindo Strings

puts()

```
puts(nome);          → Marcos da Silva
puts(&nome[4]);      → cos da Silva
```

printf()

```
printf(“%s”, nome);  → Marcos da Silva
```

Funções de Manipulação de Strings

Essas funções estão declaradas no arquivo *header* string.h, sendo necessário então a inclusão do #include <string.h> no início da implementação.

strlen()

Retorna o tamanho da string, a partir de um endereço da string até o caracter anterior a ‘\0’.

Exemplos

```
char nome[ ] = “Jose Carlos”;
strlen(nome);                → 11
strlen(&nome[2]);            → 09
```

strcat()

Concatena duas strings.
Exemplos

```
char nome[ ] = “Priscila”;
char atividade[ ] = “estuda”;
strcat(nome, atividade);          → nome = “Priscila estuda”
                                  → atividade = “estuda”
```

strcmp()

Compara duas strings. Forma de uso, strcmp(string1, string2), retorna um valor menor que 0 se string1 for menor que string2, igual a 0 se string1 for igual a string2, maior que 0 se string1 for maior que string2.
Exemplo

```
printf(“%i”, strcmp(“Ana”, “Alice”));
```

strcpy()

Copia strings.
Exemplos

```
strcpy(nome,”ANA”);
strcpy(nome1, nome2);
```

13 ESTRUTURAS

Agrupamento de um conjunto de dados não similares sob um único nome, ou seja, estruturas são tipos de variáveis que agrupam dados geralmente desiguais. Os itens de dados de uma estrutura são chamados de membros.

```
struct Estrutura
{
  tipo variável;
  .
  .
  .
};
```

Estrutura nomex; → declaração da variável nomex do tipo “Estrutura”

Por meio da palavra-chave *struct* define-se um novo tipo de dado. Definir um tipo de dado significa informar ao compilador o seu nome, tamanho em bytes e o formato em que ele deve ser armazenado e recuperado na memória.

Após ter sido definido, o novo tipo existe e pode ser utilizado para criar variáveis de modo similar a qualquer outro tipo.

Definir uma estrutura não cria nenhuma variável, somente informa ao compilador as características de um novo tipo de dado. Não há nenhuma reserva de memória.

Exemplo

```
struct TLivro
{
   int reg;
   char titulo[30];
   char autor[30];
};

TLivro livro;
```

Acessando a Estrutura

```
livro.reg = 10;
gets(livro.titulo);
strcpy(livro.autor, “Rafael de Azevedo”);
```

Matriz de Estrutura

O processo de declaração de uma matriz de estrutura é perfeitamente análago à declaração de qualquer outro tipo de matriz.

```
TLivro livro[50];

livro[0].reg = 1;
```

Inicializando uma Estrutura

```
struct TDataAniv
{
   char nome[80];
   int mês, dia, ano;
};

TDataAniv aniversario[ ] = {“Ana”, 12, 30, 73,
                            “Carlos”, 05, 13, 66,
                            “Mara”, 11, 29, 70};
```

14 PONTEIROS

É uma das mais poderosas estruturas de dados. Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente. O mecanismo usado para isto é o endereço da variável, sendo o ponteiro a representação simbólica de um endereço.

Utilização dos ponteiros:

- Manipular elementos de matrizes;
- Receber parâmetros em funções que necessitem modificar o valor original;
- Passar strings de uma função para outra;
- Criar estruturas de dados dinâmicas, como pilhas, listas e árvores, onde um item deve conter referências a outro;
- Alocar e desalocar memória do sistema.

A memória do computador é dividida em bytes, estes bytes são numerados de 0 até o limite da memória da máquina. Estes números são chamados endereços de bytes. Um endereço é a referência que o computador usa para localizar variáveis. Toda variável ocupa uma certa localização na memória, e seu endereço é o primeiro byte ocupado por ela.

O C oferece dois operadores para trabalharem com ponteiros. Um é o operador de endereço (&) que retorna o endereço de memória da variável. O segundo é o operador indireto (*) que é o complemento de (&) e retorna o conteúdo da variável localizada no endereço (ponteiro) do operando, isto é, devolve o conteúdo da variável apontada pelo operando.

Declaração

```
tipo *variável;
```

Exemplo

```
void main(void)
{
  char x = 'Z', y = 'K';
  char *px, *py;
  px = &x;
  py = &y;

  printf("\nEndereco de px: %u", &px);
  printf("\nEndereco de py: %u", &py);
  printf("\n");
  printf("\nConteudo de px: %u", px);
  printf("\nConteudo de py: %u", py);
  printf("\n");
  printf("\nConteudo pra onde px aponta: %c", *px);
  printf("\nConteudo pra onde py aponta: %c", *py);

  py++;
  printf("\nConteudo pra onde py+1 aponta: %c", *py);
}
```

Endereço	...	100	101	...	1000	1001	...
Conteúdo	...	1000	1001	...	‘Z’	‘K’	...
Variável	...	px	py	...	x	y	...

Ponteiro como Parâmetro de Função

Isto se aplica quando é necessário que a variável passada por parâmetro, após a execução da função, volte para a função principal com seu conteúdo alterado. Dá-se o nome de passagem de parâmetro por referência.

Exemplo

```
void reajusta20(float *p, float *r);

void main(void)
{
  float preco, val_reaj;
  do
  {
    printf(“Digite o preco atual: “); scanf(“%f”, &preco);
    reajusta20(&preco, &val_reaj);
    printf(“Preço novo: %f”, preço);
    printf(“\nAumento: %f”, val_reaj);
  }
  while (preco != 0.0);
}

void reajusta20(float *p, float *r)
{
  *r = *p * 0.2;
  *p = *p * 1.2;
}
```

Matriz de Ponteiro

Neste caso tem-se uma matriz e em cada posição tem um endereço de memória (ponteiro) onde se localiza um caracter. Os vetores de caracteres que formam a matriz podem ser acessados através de seus índices ou pelos ponteiros, como mostra o exemplo abaixo.

Exemplo

```
void main(void)
{
  char *semana[7]={"Domingo", "Segunda", "Terca", "Quarta", "Quinta",
                   "Sexta", "Sabado"};
  int i;
  char *aux=semana[0];

  for (i=0;semana[0][i];i++)
    printf("%c",semana[0][i]);

  printf("\n\n");
```

```
  for (i=0;*aux;i++)
  {
    printf("%c",*aux);
    aux++;
  }
}
```

Ponteiro para Estrutura

Exemplo

```
struct TLivro
{
  int reg;
  char titulo[30], autor[30];
} livro;

TLivro *pont;
pont = &livro;
pont->titulo   ou   (*pont).titulo
```

Alocação Dinâmica

Na Alocação Estática (Vetores e Matrizes) é na declaração da variável que se define o tamanho que a mesma irá ocupar na memória do computador e este valor não pode ser alterado em tempo de execução.

Ao contrário da alocação estática, na Alocação Dinâmica pode-se alterar em tempo de execução o espaço ocupado pelas variáveis na memória do computador.

É aplicada quando não se sabe o quanto será necessário de memória do computador para a resolução dos problemas, uma aplicação bastante utilizada é a lista encadeada, podendo ser dos tipos pilhas, filas ou árvores.

Exemplo

```
/* Lista Encadeada do tipo Fila */

struct TReg
{
  char nome[20];
  TReg *prox;
};

void inserir(TReg **inic, TReg **fim);
void remover(TReg **inic, TReg **fim);
```

```
void main(void)
{
  TReg *inic=NULL, *fim=NULL;
  char op='1';
  while (op!='3')
  {
    clrscr();
    printf("\n1 - Inserir");
    printf("\n2 - Remover");
    printf("\n3 - Finalizar");
    printf("\nOpcao: "); op=toupper(getche());
    switch (op)
    {
      case '1':
        inserir(&inic,&fim);
        break;
      case '2':
        remover(&inic,&fim);
        break;
    }
  }
}

void inserir(TReg **inic, TReg **fim)
{
  TReg *p, *aux;
  p=new(TReg);
  printf("\nNome: "); gets(p->nome);
  p->prox=NULL;
  if (*fim)
  {
    aux=*fim; aux->prox=p;
  }
  *fim=p;
  if (!*inic) *inic=p;
}

void remover(TReg **inic, TReg **fim)
{
  TReg *p;
  if (*inic)
  {
    p=*inic;
    printf("\n%s removido.",p->nome);
    *inic=p->prox;
    if (!*inic) *fim=NULL;
    delete p;
    getche();
  }
}
```

15 ENTRADA/SAIDA PELO ARQUIVO

Streams e Arquivos

O sistema de E/S de C fornece uma interface consistente ao programador C, independentemente do dispositivo real que é acessado. Isto é, o sistema de E/S provê um nível de abstração entre o programador e o dispositivo utilizado. Essa abstração é chamada de *stream* e o dispositivo real é chamado de arquivo.

Streams

O sistema de arquivos de C é projetado para trabalhar com uma ampla variedade de dispositivos, incluindo terminais, acionadores de disco e acionadores de fita.

Embora cada um dos dispositivos seja muito diferente, o sistema de arquivos com *buffer* transforma-os em um dispositivo lógico chamado de *stream*.

Arquivos

Em C, um arquivo pode ser qualquer coisa, desde um arquivo em disco até um terminal ou uma impressora. Você associa uma *stream* com um arquivo específico realizando uma operação de abertura. Uma vez o arquivo aberto, informações podem ser trocadas entre ele e o seu programa.

As Funções mais Comuns

Nome	Função
fopen()	Abre um arquivo
fclose()	Fecha um arquivo
fputc()	Escreve um caracter em um arquivo
fgetc()	Lê um caracter de um arquivo
fputs()	Escreve uma string em um arquivo
fgets()	Lê uma string de um arquivo
fprintf()	É para um arquivo o que printf() é para o console
fscanf()	É para um arquivo o que scanf() é para o console
fwrite()	Escreve tipos de dados maiores que um byte em arquivo
fread()	Lê tipos de dados maiores que um byte em arquivo
feof()	Devolve verdadeiro se o fim de arquivo for atingido
ferror()	Devolve verdadeiro se ocorreu um erro
remove()	Apaga um arquivo
fseek()	Posiciona no arquivo em um byte específico

Modos de abertura de arquivo

Modo	Significado
r	Abre um arquivo texto para leitura
w	Cria um arquivo texto para escrita
rb	Abre um arquivo binário para leitura
wb	Cria um arquivo binário para escrita
r+	Abre um arquivo texto para leitura e escrita
w+	Cria um arquivo texto para leitura e escrita
rb+	Abre um arquivo binário para leitura e escrita
wb+	Cria um arquivo binário para leitura e escrita

Usando fopen(), fgetc(), fputc() e fclose()

Exemplo

```
/* Do teclado para o disco */

void main(void)
{
  FILE *fp;
  char ch;
  fp = fopen("texto.txt", "w");
  if (fp == NULL)
  {
    printf("O arquivo não pode ser criado");
    exit(1);
  }
  do {
    ch = getchar();
    fputc(ch, fp);
  } while (ch != '$');
  fclose(fp);
}
```

Trabalhando com Strings: fputs() e fgets()

Exemplo

```
void main(void)
{
  FILE *fp;
  char str[80];
  fp = fopen("texto2.txt", "w");
  if (fp == NULL)
  {
    printf("O arquivo não pode ser criado");
    exit(1);
  }
```

```
    do {
      printf("Digite uma string (Enter para sair):\n "); gets(str);
      strcat(str, "\n");
      fputs(str, fp);
    } while (*str != '\n');
    fclose(fp);
}
```

Usando fread() e fwrite()

Estas duas funções são específicas para leitura e escrita de tipos de dados mais complexos (maiores que um byte), como float, int ou estruturas.

Exemplo

```
/* Uma funcao para ler e outra para gravar uma matriz de
    estruturas em arquivo */

struct TReg {
  char nome[40];
  char endereco[40];
  char cidade[30];
  char estado[03];
  char cep[10];
} Registro[50];

/* Salva a matriz */
void salva(void)
{
  FILE *fp;
  int i;

  if ((fp=fopen("Registro.dat", "wb"))==NULL)
  {
    printf("O arquivo não pode ser criado.");
    return;
  }

  for (i=0; i<50; i++)
    if (*Registro[i].nome)
      if (fwrite(&Registro[i], sizeof(TReg), 1, fp)!=1)
        printf("Erro de escrita no arquivo.");

  fclose(fp);
}
```

```
/* Le a matriz */
void le(void)
{
  FILE *fp;
  int i;

  if ((fp=fopen("Registro.dat", "rb"))==NULL)
  {
    printf("O arquivo não pode ser aberto.");
    return;
  }

  inicializa();
  for (i=0; i<50; i++)
    if (fread(&Registro[i], sizeof(TReg), 1, fp)!=1)
    {
      if (feof(fp)) break;
      printf("Erro de leitura no arquivo.");
    }

  fclose(fp);
}


/* Inicializacao da matriz – nome[0] = '\0' */
void inicializa(void)
{
  int t;
  for (t=0; i<50; t++) *Registro[t].nome = '\0';
}
```

BIBLIOGRAFIA

SCHILDT, Herbert. **C Completo e Total.** Makron Books, São Paulo, 1996.

WIENER, Richard. **Turbo C, passo a passo.** Campus, Rio de Janeiro, 1991.