7. MATRIZES E *STRINGS*

7.1. Matrizes

Uma matriz é uma coleção de variáveis do mesmo tipo que é referenciada por um nome comum. Um elemento específico em uma matriz é acessado por meio de um índice. Em C, todas as matrizes consistem em posições contíguas na memória, e toda matriz tem 0 como o índice do seu primeiro elemento. O endereço mais baixo corresponde ao primeiro elemento e o mais alto, ao último elemento. Matrizes podem ter de uma a várias dimensões.

7.1.1. Matrizes Unidimensionais

Forma geral: *<tipo> <nome_var>[<tamanho>];*

Exemplo:

```
main() {
        int x[100];         // reserva 100 elementos inteiros
        int t;
        for (t=0; t<100; ++t) x[t] = t;
}
```

A quantidade de armazenamento necessário para guardar uma matriz está diretamente relacionada com seu tamanho e seu tipo (total em *bytes* = *sizeof*(tipo) * tamanho da matriz). C não tem verificação de limites em matrizes (o compilador não acusa este tipo de erro), o programador é quem deve prover verificação dos limites onde for necessário.

É possível gerar um ponteiro para o primeiro elemento de uma matriz simplesmente especificando o nome da matriz, sem nenhum índice. Além disso, o primeiro elemento de uma matriz também pode ser especificado através do operador &. Exemplo:

```
main() {
        int sample[10];         // reserva 10 elementos inteiros
        int *p;                 // ponteiro para inteiro
        p = sample;             // ponteiro p recebe o ponteiro do primeiro elemento da matriz (mais comum)
        p = &sample[0];         // ponteiro p recebe o ponteiro do primeiro elemento da matriz (não usado)
}
```

Conforme já visto na seção 4.3. Passagem de Parâmetro, não é possível passar uma matriz inteira como um argumento para uma função. É passado apenas um ponteiro para uma matriz. No exemplo a seguir é possível verificar que o comprimento da matriz não importa à função, porque C não realiza verificação de limites [SCH 96].

```
main() {
        int i[10];
        func1(i);
}
void func1(int *x)          //Ponteiro
{ ... }
void func1(int x[10])       //Matriz dimensionada
{ ... }
void func1(int x[])         //Matriz não-dimensionada
{ ... }
```

7.1.2. Matrizes Bidimensionais

Uma matriz bidimensional, para inteiros por exemplo, de tamanho 10,20 é declarada da seguinte maneira:

```
int d[10][20];
```

Similarmente, para acessar, por exemplo, o ponto 1,2 da matriz d, deve-se utilizar:

```
d[1][2];
```

O exemplo a seguir carrega uma matriz bidimensional com os números de 1 a 12 e escreve-os linha por linha.

```
#include <stdio.h>
main() {
  int t, i, num[3][4];

  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;

  // Mostra os números
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i)
      printf("%3d ", num[t][i]);
    printf("\n");
  }
}
```

Neste exemplo, "num[0][0]" tem o valor 1, "num[0][1]" tem o valor 2, "num[0][2]" o valor 3 e assim por diante. O valor de "num[2][3]" será 12. A matriz "num" pode ser visualizada na figura 7.1.

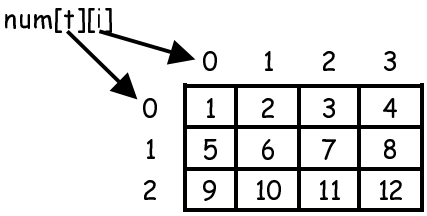

Figura 7.1 – Matriz bidimensional

Matrizes bidimensionais são armazenadas em uma matriz linha-coluna, onde o primeiro índice indica a linha e o segundo a coluna. Isso significa que o índice mais à direita varia mais rapidamente do que o mais à esquerda quando os elementos da matriz são acessados na ordem em que estão realmente armazenados na memória.

No caso de uma matriz bidimensional, a seguinte fórmula fornece o número de *bytes* de memória necessários para armazená-la:

Bytes = tamanho do 1º índice * tamanho do 2º índice * *sizeof* (tipo)

Portanto, assumindo inteiros de dois *bytes*, uma matriz de inteiros com dimensões 10,5 teria 10*5*2 *bytes* alocados.

Quando uma matriz bidimensional é usada como um argumento para uma função, apenas um ponteiro para o primeiro elemento é realmente passado. Porém, uma função que recebe uma matriz bidimensional como um parâmetro deve definir pelo menos o comprimento da segunda dimensão. Isso ocorre porque o compilador C precisa saber o comprimento de cada linha para indexar a matriz corretamente. Por exemplo, uma função que recebe uma matriz bidimensional de inteiros com dimensões 10,10 é declarada desta forma:

```
void func1(int x[][10]) { ... }
```

É possível especificar a primeira dimensão, se quiser, mas não é necessário. O compilador C precisa saber a segunda dimensão para trabalhar em sentenças como "x[2][4]" dentro da função. Se o comprimento das linhas não é conhecido, o compilador não pode determinar onde a terceira linha começa.

O próximo programa usa uma matriz bidimensional para armazenar as notas numéricas de cada aluno de uma sala de aula. O programa assume que o professor tem três turmas e um máximo de 30 alunos por turma. Note a maneira como a matriz "grade" é acessada em cada uma das funções [SCH 96].

```
// Um banco de dados simples para notas de alunos
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define CLASSES  3                  // Constante
#define GRADES  30                  // Constante
int grade [CLASSES][GRADES];        // Variável global
void enter_grades(void);            // Protótipo
int get_grade(int num);             // Protótipo
void disp_grades(int g[][GRADES]);  // Protótipo

// Função principal
main() {
  char ch, str[80];
  for(;;) {
    do {
      printf("(E)ntrar notas\n");
      printf("(M)ostrar notas\n");
      printf("(S)air\n");
      gets(str);
      ch = toupper(*str);
    } while(ch!='E' && ch!='M' && ch!='S');
    switch(ch) {
      case 'E':
            enter_grades();
            break;
      case 'M':
            disp_grades(grade);
            break;
      case 'S':
            exit(0);
    }
  }
}

// Função para entrada de notas dos alunos.
void enter_grades(void){
  int t, i;
  for(t=0; t<CLASSES; t++) {
    printf("Turma # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade[t][i]= get_grade(i);
  }
}

// Função para leitura de uma nota.
int get_grade(int num) {
  char s[80];
  printf("entre a nota do aluno # %d:\n",num+1);
  gets(s);
  return(atoi(s));
}
```

```
// Função para mostrar as notas.
void disp_grades(int g[][GRADES]) {
  int t, i;
  for(t=0; t<CLASSES; ++t) {
    printf("Turma # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      printf("aluno #%d e %d\n", i+1, g[t][i]);
  }
}
```

7.1.3. Matrizes Multidimensionais

C permite matrizes com mais de duas dimensões. O limite exato, se existe, é determinado pelo compilador. A forma geral da declaração de uma matriz multidimensional é:
<tipo> <nome> [tamanho1] [tamanho2] [tamanho3] ... [tamanhoN];

Matrizes de três ou mais dimensões não são freqüentemente usadas devido à quantidade de memória que ocupam, uma vez que o armazenamento necessário cresce exponencialmente com o número de dimensões. Além disso, em matrizes multidimensionais toma-se tempo do computador para calcular cada índice. Isso significa que acessar um elemento em uma matriz multidimensional é mais lento do que acessar um elemento em uma matriz unidimensional.

Para passar matrizes multidimensionais para funções deve-se declarar todas menos a primeira dimensão. Exemplo:

```
main() {
        int  m[4][3][6][5];
        func1(m);
}
void func1(int d[][3][6][5]) { ... }
```

7.2. Strings

Strings em C consiste em uma matriz unidimensional de caracteres terminada por nulo ('\0'). Por isso, é preciso declarar matrizes de caracteres como sendo um caractere mais longo que a maior *string* que elas devem guardar. Por exemplo, a declaração de uma matriz "str" que guarda uma *string* de 10 caracteres, deve ser: *char str[11]; // um espaço a mais para o nulo*

Embora C não tenha o tipo de dado *string*, ela permite constantes *string*. Uma "constante *string*" é uma lista de caracteres entre aspas. Por exemplo: "Alo Mundo!". Não é necessário adicionar nulo no final das constantes *string* manualmente, pois o compilador C faz isso automaticamente.

C suporta uma ampla gama de funções de manipulação de *strings*. As mais comuns são:

Nome	Função
strcpy(s1, s2)	Copia s2 em s1.
strcat(s1, s2)	Concatena s2 ao final de s1.
strlen(s1)	Retorna o tamanho de s1.
strcmp(s1, s2)	Retorna 0 se s1 e s2 são iguais; menor que 0 se s1<s2; maior que 0 se s1>s2.
strchr (s1, ch)	Retorna um ponteiro para a primeira ocorrência de ch em s1.
strstr(s1, s2)	Retorna um ponteiro para a primeira ocorrência de s2 em s1.

Essas funções usam o cabeçalho padrão "string.h". O próximo programa ilustra o uso dessas funções de *string*:

```
#include <stdio.h>
#include <string.h>
main() {
  char s1[80], s2[80];
  gets(s1);
  gets(s2);

  printf("Comprimentos: %d %d\n", strlen(s1), strlen(s2));

  if(!strcmp(s1, s2)) printf ("As strings sao iguais \n");

  strcat(s1, s2);
  printf("%s \n", s1);

  strcpy(s1, "Isto eh um teste.\n");
  printf(s1);
  if(strchr("alo", 'o')) printf("o esta em alo\n");
  if(strstr("ola  aqui", "ola")) printf("ola encontrado");
}
```

Se esse programa for executado e se for digitado "alo" e "alo", a saída será:

```
comprimentos: 3 3
As string são iguais
aloalo
Isso eh um teste.
o esta em alo
ola encontrado
```

É possível que surja a necessidade de usar uma matriz de *strings*, que é criada como uma matriz bidimensional de caracteres. Neste caso, o tamanho do índice esquerdo indica o número de *strings* e o tamanho do índice do lado direito especifica o comprimento máximo de cada *string*. O exemplo a seguir declara uma matriz de 30 *strings*, cada qual com um comprimento máximo de 79 caracteres.

```
char str_array[30][80];
```

É fácil acessar uma *string* individual, basta especificar apenas o índice esquerdo. Por exemplo, os seguintes comandos chamam *gets()* com a terceira *string* em "str_array":

```
gets(str_array[2]);         // mais usual
get(&str_array[2][0]);      // equivalente ao comando anterior
```

O próximo exemplo, que usa uma matriz de *string* como base para um editor de texto muito simples, ilustra como matrizes de *string* funcionam [SCH 96].

```
#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];

void main(void) {
  register int t, i, j;

  printf("Entre com uma linha vazia para sair.\n");
  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* sai com linha em branco */
  }
  for(i=0; i<t; i++) {
   for(j=0; text[i][j]; j++) putchar(text[i][j]);
   putchar('\n');
  }
}
```

7.3. Inicialização de Matrizes

C permite a inicialização de matrizes no momento da declaração. A forma geral de uma inicialização de matriz é semelhante à de outras variáveis:
<especificador_tipo> <nome> [tamanho1] [tamanho2] [tamanho3] ... [tamanhoN] = {lista_de_valores};
A lista de valores é uma lista separada por vírgulas de constantes cujo tipo é compatível com "<especificador_tipo>". A primeira constante é colocada na primeira posição da matriz, a segunda na segunda posição, e assim por diante.

No exemplo seguinte, uma matriz inteira de dez elementos é inicializada com os números de 1 a 10, isto é, "i[0]" terá o valor 1 e "i[9]" terá o valor 10:
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

Matrizes multidimensionais são inicializadas da mesma forma que matrizes unidimensionais. Por exemplo:

```
int sqrs[10][2] = {
        1, 1,
        2, 4,
        3, 9,
        4, 16,
        5, 25,
        6, 36,
        7, 49,
        8, 64,
        9, 81,
        10, 100
};
```

Matrizes de caracteres que contêm *strings* permitem uma inicialização abreviada que toma a forma:
char nome_da_matriz[tamanho] = "string";
char str[14] = "Eu gosto de C";
A declaração e inicialização de "str" também poderia ser feita da seguinte maneira:
char str[14] = {'E', 'u', ' ', 'g', 'o', 's', 't', 'o', ' ', 'd', 'e', ' ', 'C', '\0'};
Como todas as *strings* em C terminam com um nulo, deve-se garantir que a matriz seja longa o bastante para incluir o nulo. Por isso "str" tem comprimento de 14 caracteres, embora "Eu gosto de C" tenha apenas 13. Quando uma constante *string* é usada, o compilador automaticamente fornece o terminador nulo [SCH 96].