ESc 101: Fundamentals of Computing

Lecture 34

Apr 5, 2010

Making Matrix Size Variable

- As for large numbers, we can make the size of a matrix also variable using `malloc()`.
- We can associate two variables with a matrix: `num_rows` and `num_cols` representing the number of rows and columns of the matrix.
- These variables are given values during execution and then space for the matrix is allocated.

Creating Two Dimensional Arrays

- A single dimensional array of floats of variable size `s` can be created by:
  `malloc( sizeof(float) * s );`
- Function `sizeof()` takes as input a type name.
- It returns the size required to store a variable of that type.
- For example:

```
sizeof(int) = 4
sizeof(char) = 1
sizeof(float) = 4
sizeof(float *) = 8
```

Creating Two Dimensional Arrays

- To create a two dimensional array of floats of size s × t, we first create a single dimensional array of pointers:
  `mat = (float **) malloc( sizeof(float *) * s );`
- In above, mat points to the first element of this array of pointers.
- Now, for each element of this array, we create a single dimensional array of t floats:
  `mat[i] = (float *) malloc( sizeof(float) * t );`
- mat[i][j] is the jth number of the ith row.

Function allocate_matrix()

```
typedef float **Matrix;

Matrix allocate_matrix(int num_rows, int num_cols)
{
    Matrix mat;

    // create an array of num_rows pointers,
    // and make mat point to it.
    mat = (Matrix) malloc( sizeof(float *) * num_rows );

    for (int i = 0; i < num_rows; i++)
       // create an array of num_cols floats,
       // and make mat[i] point to it.
       mat[i] = (float *) malloc( sizeof(float) * num_cols );

    return mat;
}
```

handling Matrices of Different Size

- To make the library even more useful, we should allow matrices of different sizes to be created simultaneously.
- This may be required, for example, in doing vector algebra and matrix multiplication.
- This means that for every matrix, two size parameters are to be associated.
- Defining three variables for every matrix is very cumbersome though.
- C provides the facility to group them together using `struct` command.

struct Command

The format of the command is:

```
struct <name>
    type1 <field1>;
    type2 <field2>;
    ...
    typem <fieldm>;
```

This defines <name> to represent a collection of parameters <field1>, ..., <fieldm> of type1, ..., typem respectively.

Using `struct`

- We can now define variables of type `struct <name>`:
  `struct <name> <var>;`
- variable `<var>` is defined and space is allocated for all its fields.
- The fields are accessed as: `<var>.<field1>`, . . ., `<var>.<fieldm>`.

Defining Matrix Type

```
// structure to store matrices
struct matrix {
    int rows; // number of rows
    int cols; // number of columns
    float **element; // pointer to elements
}

struct matrix mat; // variable of type struct matrix
```

Function allocate_matrix() Again

```
struct matrix allocate_matrix(int n_rows, int n_cols)
{
    struct matrix mat;

    mat.rows = n_rows;
    mat.cols = n_cols;
    // create an array of n_rows pointers,
    // and make mat.element point to it.
    mat.element = (float **) malloc(sizeof(float *) * n_rows);

    for (int i = 0; i < num_rows; i++)
       // create an array of n_cols floats,
       // and make mat.element[i] point to it.
       mat.element[i] = (float *) malloc(sizeof(float) * n_col

    return mat;
}
```

Redefining Type Matrix

Define Matrix as:

```
typedef struct matrix Matrix;
```

Or, do it directly as:

```
typedef struct matrix {
    int rows; // number of rows
    int cols; // number of columns
    float **element; // pointer to elements
} Matrix;
```