Subject #13: Arrays

- Type the following program, compile it and run it

  ```
  #include <stdio.h>

  #define MAXNUMS 5

  main()
  {
       int nums[MAXNUMS]; /* Define an array of MAXNUMS integers */
       int i;

       printf("Enter %d numbers:\n", MAXNUMS);
       for (i=0; i<MAXNUMS; i++)
            scanf("%d", &nums[i]);

       printf("The numbers in reveresed order:\n");
       for (i=MAXNUMS-1; i>=0; i--)
            printf("%d ", nums[i]);
       printf("\n");
  }
  ```

- The variable **nums** is an *array* of MAXNUMS (that is, five) integers. The compiler allocates enough consecutive memory space, and then for each $0 \leq i <$ MAXNUMS, **nums[i]** is just like an *int*. There can be arrays of any type.
- The compiler must know the size of an array before the program is executed, so MAXNUMS cannot be a variable, and we don't want to enter "magic numbers" into the program. Therefore, we use a symbolic constant. Notice that by its use, the program is clearer, and if we decide to change the number of numbers, we can do it very easily.
- Array subscripts *always* start at 0. The largest subscript allowed, therefore, is one less than the array size. It is a grave (and common) bug to exceed the allowed range of subscripts.
- A string is just an array of *char*s, with its end specified by the special character **'\0'**. The string **"hello"** is thus six characters long and requires an array of at least six characters. Similarly, the string constant **"hello\n"** will be created as an array of seven characters. Do you see why? Remember that '\n' is a single character!
- Naturally, the array should be initialized or else it will contain garbage.
- Arrays can be initialized in their declaration. For example,

  ```
  int numbers[4] = {4,3,2,1} ;
  ```

- Unlike ordinary parameters, which are passed to functions by value and any changes made to them inside functions are not reflected outside, changes made to arrays passed to a function as parameters *are* retained outside of the function.

- Arrays can be multi-dimensional. Suppose we want to read grades of an unknown number of students, when for each student there are the grades of four courses in a line. Then we should have a symbolic constant MAX_STUDENTS defined, and the grades array will be declared as

  ```
  int grades[MAX_STUDENTS][4] ;
  ```

  A function to read the grades can be the following (without much of error checking):

  ```
  int read_grades(FILE *infp, int grades[][4])
  {
      int st = 0 ;

      while (fscanf(infp, "%d %d %d %d", &grades[st][0], &grades[st][1],
                                         &grades[st][2], &grades[st][3] ) != EOF)

          st++ ;
      return st ;
  }
  ```

  This function may be called as follows:

  ```
  read_grades(infp, grades) ;
  ```

- Multi-dimensional arrays are represented in memory just like one-dimensional ones — in consecutive addresses. The "leftmost" dimension (that of size MAX_STUDENTS in our case), is the "slowest to run." Therefore, in order for functions to know how to handle their array parameters, they must be supplied with the sizes of all of their dimensions, with the possible exception of the "leftmost" one.
- The following program reads a list of students' grades from the standard input, and prints their histogram (but in lines and not in columns):

  ```
  #include <stdio.h>
  #define NUMLINES 20
  #define GRADES_IN_LINE ( (100+(NUMLINES)-1) / (NUMLINES) )

  main()
  {
      int count[NUMLINES] ; /* The no. of students in each grade range */
      int i, j, grade, line ;

      for (i = 0 ; i < NUMLINES ; i++)
          count[i] = 0 ;

      while (scanf("%d", &grade) != EOF) {
          if (grade < 0  ||  grade > 100) {
              /* Validate the input */
              fprintf(stderr, "grade %d is illegal\n", grade) ;
  ```

```
                continue ;
            }
            if (grade == 0)
                line = 0 ; /* Make sure that line=0 in this case */
            else
                line = (grade-1) / GRADES_IN_LINE ; /* Compute the right line */
            count[line]++ ;
        }

        for (i = 0 ; i < NUMLINES ; i++) {
            printf("|") ;
            for (j = 0 ; j < count[i] ; j++)
                printf("*") ; /* Print a count[i] long line of asterisks */
            printf("\n") ;
        }
    }
```

- When input is read from the user through the keyboard, EOF is entered by pressing Ctrl-d.
- Notice the care we took in defining GRADES_IN_LINE, in checking the admissibility of **grade**, and in dealing with the grade 0 — in order to avoid bugs from exceeding the allowed index range of the array.

Subject #14: Structures

- Structures allow you to keep several related pieces of information in a single variable.
- Structures are declared by specifying the items they contain. Each item is a variable of a certain type. The following declaration defines a structure called *student* that can be used to describe students in the course "Computers for physicists":

```
struct student {
    char name[50];
    int id;
    int ex_grades[5];
    int project_grade;
    int exam_grade;
    int final_grade;
};
```

- The line:

```
struct student s;
```

  declares that s is a structure of type *student*. The items of s can be referenced as in the following example:

```
s.final_grade = (s.project_grade + s.exam_grade)/2;
```

- If s1 and s2 are structures of the same type, we may write:

  ```
  s1 = s2
  ```

  This will assign to each item in s1 the value that it has in s2. You cannot compare structures but only their individual items:

  ```
  if (s1 == s2)        – not allowed
  if (s1.id == s2.id)  – allowed
  ```

- You can also create arrays of structures. For example, suppose we have the following definitions:

  ```
  struct student students[50]; // Students in the course
  int n;                       // Number of students
  int sum;
  float average;
  int i;
  ```

  The following lines will calculate the average grade in the course:

  ```
  sum = 0;
  for (i = 0; i < n; i++)
      sum += students[i].final_grade;
  average = (float)sum/n;
  ```

- Structures, as any other kind of variable, may be passed as arguments to functions. Here is an example:

  ```
  void summarize_student(struct student s)
  {
      s.final_grade = (s.exam_grade + s.project_grade)/2;
      printf("Name: %s, ID: %d, Grade: %d\n",
                    s.name, s.id, s.final_grade);
  }
  ```

  The function call would look like this:

  ```
        struct student st;

            .
            .
            .

        summarize_student(st);
  ```

  Notice that `s.final_grade` does not retain its value outside the function.

Classwork: Multiplying a vector by a matrix.

- We shall write a function to multiply a vector by a square matrix. They will be declared of sizes `MAX` and `MAX` × `MAX` respectively, with `MAX` a symbolic constant (using `#define`), but their actual sizes can be smaller.
- Write a function `void mult(int n, float matrix[MAX][MAX], float vector[MAX], float result[MAX])` multiplying an `n` × `n` matrix by an `n`-element vector.
- Write a program that checks the function.

Homework #6: Bubble Sort

- There are many ways to sort a list of numbers. The algorithm we will use is called *Bubble Sort*.
- The heart of the algorithm is a loop over all the nearest neighbours {vec(i),vec(i+1)} to check that {vec(i)≤ vec(i+1)} and replace them if not.
- This loop is repeated until the list is sorted.
- Example:

$$\text{first loop: }\left\{\begin{matrix}6\\5\\1\\4\end{matrix}\right\}\Rightarrow\left\{\begin{matrix}5\\6\\1\\4\end{matrix}\right\}\Rightarrow\left\{\begin{matrix}5\\1\\6\\4\end{matrix}\right\}\Rightarrow\left\{\begin{matrix}5\\1\\4\\6\end{matrix}\right\}$$

$$\text{second loop:}\left\{\begin{matrix}5\\1\\4\\6\end{matrix}\right\}\Rightarrow\left\{\begin{matrix}1\\5\\4\\6\end{matrix}\right\}\Rightarrow\left\{\begin{matrix}1\\4\\5\\6\end{matrix}\right\}$$

- **Homework:** Write a program that sorts a list of (integer) numbers.
  - Write a function that sorts an array. The array and its size are given to the function as parameters.
  - The *main* should read the input from a file called **sort.in**, print it, sort it, and print it again.

Good luck