CS11001/CS11002 Programming and Data Structures (PDS)

(Theory: 3-1-0)

Introduction to arrays

What are Arrays?

- Arrays are our first example of *structured data*.
- Think of a book with pages numbered 1,2,...,400.
- The book is a single entity, has its individual name, author(s), publisher, etc. but the contents of its different pages are (normally) different.
- Moreover, Page 251 of the book refers to a particular page of the book.
- To sum up, individual pages retain their identities and still we have a special handy bound structure treated as a single entity..

A motivating example

- Now imagine that you plan to sum 400 integers.
- Where will you store the individual integers? Thanks to your ability to declare variables, you can certainly do that.
- Declare 400 variables with 400 different names, initialize them individually and finally add each variable separately to an accumulating sum. That's gigantic code just for a small task.
- Arrays are there to help you. Like your book you now have a single name for an entire collection of 400 integers.
- Declaration is small.
- Codes for initialization and addition also become shorter, because you can now access the different elements of the collection by a unique index.
- There are built-in C constructs that allow you do parameterized (i.e., indexed) tasks repetitively.

Declaring arrays

- Similar to that of declaring data types.
- For example, the declaration
  - int intHerd[400]; creates an array of name intHerd that is capable of storing 400 int data.
- A more stylistic way to do the same is illustrated now.
  - #define HERD_SIZE 400
    int intHerd[HERD_SIZE];
    Note that all individual elements of a single array must be of the same type. You cannot declare an array some of whose elements are integers, the rest floating-point numbers. Such heterogeneous collections can be defined by other means that we will introduce later.

Accessing individual array elements

- Once an array A of size s is declared, its individual elements are accessed as A[0],A[1],...,A[s-1].
- It is very important to note that: <u>Array indexing in C is zero-based.</u>
- This means that the "first" element of A is named as A[0] (not A[1]), the "second" as A[1], and so on. The last element is A[s-1].
- Each element A[i] is of data type as provided in the declaration. For example, if the declaration goes as:
- int A[32]; each of the elements A[0],A[1],...,A[31] is a *variable* of type int.
- You can do on each A[i] whatever you are allowed to do on a single int variable.

C does not provide automatic range checking.

- If an array A of size s is declared, the element A[i] belongs to the array (more correctly, to the memory locations allocated to A) if and only if 0 <= i <= s-1.
- However, you can use A[i] for other values of i.
- No compilation errors (nor warnings) are generated for that. Now when you run the program, the executable attempts to access a part of the memory that is not allocated to your array, nor perhaps to (the data area allocated to) your program at all.
- You simply do not know what resides in that part of the memory. Moreover, illegal memory access may lead to the deadly "segmentation fault".
  - C is too cruel at certain points. Beware of that!

Initializing an Array

- Arrays can be initialized during declaration.
- For that you have to specify constant values for its elements.
- The list of initializing values should be enclosed in curly braces. For the declaration
  - int A[5] = { 51, 29, 0, -34, 67 }; A[0] is initialized to 51, A[1] to 29, A[2] to 0, A[3] to -34 and A[4] to 67.
  - Similarly, for the declaration char C[10] = { ‘k', ‘h', ‘a', ‘r', ‘a', g', ‘p', ‘u’, ‘r’, '\0' };
    - C[0] gets the value ‘k', C[1] the value ‘h', and so on. The last (10th) location receives the null character.
    - Such null-terminated character arrays are also called **strings**. Strings can be initialized in an alternative way. The last declaration is equivalent to: char C[10] = “kharagpur";
    - Now see that the trailing null character is missing here. C automatically puts it at the end.
  - Note also that for individual characters, C uses single quotes, whereas for strings, it uses double quotes.

Initializing an Array

- If you do not mention sufficiently many initial values to populate an entire array, C uses your incomplete list to initialize the array locations at the lower end (starting from 0).
- The remaining locations are initialized to zero. For example, the initialization
  - int A[5] = { 51, 29 }; is equivalent to int A[5] = { 51, 29, 0, 0, 0 }; If you specify an initialization list, you may omit the size of the array. In that case, the array will be allocated exactly as much space as is necessary to accommodate the initialization list.
  - You must, however, provide the square brackets to indicate that you are declaring an array; the size may be missing between them.
- int A[] = { 51, 29 }; creates an array A of size 2 with A[0] holding the value 51 and A[1] the value 29. This declaration is equivalent to

  int A[2] = { 51, 29 }; but not to int A[5] = { 51, 29 };

An example to find the largest and smallest element in the vector

```
#include<stdio.h>
main()
{
  int i, n;
  float a[50], large, small;
  printf("Size of vector? ");
  scanf("%d",&n);
  printf("\n Vector elements?\n");
  for(i=0;i<n;i++) scanf("%f",&a[i]);
```

An example to find the largest and smallest element in the vector

```
large=a[0]; small=a[0];
for(i=1;i<n;i++)
{
  if(a[i]>large)
     large=a[i];
  else if(a[i] < small)
     small=a[i];
}
printf("\n Largest element in vector is % f\n",large);
printf("\n Smallest element in vector is %f\n",small);
}
```

Name of the array

- Take the case where f is an array of floating point numbers. It is declared as:
  - float f[50];
- The name of the array is f
  - f denotes the memory location of the first floating point number in the array.
  - Thus f is equivalent to &f[0].
- The statement

```
                    scanf("%f",&f[0]);
```

is thus equivalent to

```
                    scanf("%f",f);
```

Passing arrays to functions

- We have seen how individual values (variables and pointers) can be passed to functions. Now let us see how we can pass an entire array to a function.
- Suppose an array is defined as:

```
#define MAXSIZE 100
int myarr[MAXSIZE];
```

In order to pass the array myarr to a function **foonction** one may define the function as:

```
int foonction ( int A[MAXSIZE] , int size ) { ... }
```

Function call

- This function takes two arguments, the first is an array of size MAXSIZE, and the second an integer argument named size.
- Here this second argument is meant for passing the *actual* size of the array.
- Your array can hold 100 integers. However, at a certain point of time you may be using only 32 locations (0 through 31) of the array. The other unused locations also hold some values. If they are not initialized, they contain unpredictable values.
- You do not want these garbage values to be interpreted by your function as important ones.
- So you specify the actual size to be 32. The function call should go like this:

```
foonction(myarr,32);
```

Write a function to sort n integers in ascending order: Bubble Sort

```
#include<stdio.h>
main()
{
 int a[20], i, n;
 void sort_it(int a[], int );// also void sort_it(int [], int) is correct
 printf("Enter the number of elements in the array (less than 21):");
 scanf("%d",&n);
 printf("Enter the elements\n");
 for(i=0;i<n;i++)
   scanf("%d",&a[i]);
 sort_it(a,n);
 printf("The sorted array\n");
 for(i=0;i<n;i++)
  printf("%d ",a[i]);
 printf("\n");

}
```

The sort function

```
void sort_it(int a[],int n)
{
 int i=0, j, f=0, temp;

 for(i=0;i<n;i++)
  for(j=i;j<n;j++)
    if(a[i]>a[j])
    {
     temp=a[i];
     a[i]=a[j];
     a[j]=temp;
    }
}
```

•This is an example of “call by reference.
Any change in the array in the function also changes corresponding values in the calling function.
•This is because like the swap function, here we are passing the address of the 0th location of the address.

Call by reference

- In the function definition, you need not write a[20]
- This is because, what is passed is the value of the 0th address of the array (called base address) in the calling function (here main).
- All the subsequent access to the array is being done by adding the index of the array location to the base address.
- Thus any modifications done in the function are reflected in the calling function.