Arrays

Arrays

- A sequence of elements of same type which share a single name.
- Two categories of arrays are there
  - 1. Static arrays
  - 2. Dynamic arrays.
- Declaration for static array specifies the array size, which cannot be altered afterwards.
- But for dynamic arrays the size can be altered.
  - There is something called as dynamic memory using which dynamic array's size can be modified.
  - This can be discussed only after pointers.

Initializing arrays (for static arrays)

- int n[4] = { 10, 20, 30, 40};
  - n[0] = 10, n[1] = 20, n[2] = 30, n[3] = 40
- int n[4] = { 10, 20};
  - n[0] = 10, n[1] = 20, n[2] = 0, n[3] = 0
- If initialization is done (which implicitly specifies the size) then one can omit size.
- int a[ ] = { 0, 1, 2};
  - a is of size 3 elements and a[0] = 0, a[1] = 1, a[2] = 2

Initializing character arrays

- char str[ ] = “cat”;
- This is equivalent to

  char str[ ] = {‘c’, ‘a’, ‘t’, ‘\0’};

  which is in turn equivalent to

  char str[4] = {‘c’, ‘a’, ‘t’, ‘\0’};

arrays

- int n[10] = {0};
- First element of n[ ] is explicitly initialized to 0, and the remaining elements by default are initialized to 0.
- int n[10];
- In this all 10 elements contains junk values.
- Forgetting to initialize the elements of an array whose elements should be initialized is a common mistake.
- int a[3] = {1,2,3,4,5}; /* syntax error */

arrays

Let the number of students are increased to 66

```
#include<stdio.h>
main( )
{
   int marks[54];

   /* to find average */
   for( j = 0; j < 54; j++ ){
   ……;
   }

   ……;
}
```

This should be modified from 54 to 66

This also should be modified.

Like this if at many places 54 is hard coded, then they all should be modified to avoid errors. *This is a potential problem*.

Symbolic constants

```
#include<stdio.h>
#define SIZE 54
main( )
{
   int marks[SIZE];

   /* to find average */
   for( j = 0; j < SIZE; j++ ){
   ......;
   }

   ......;
}
```

#define is a preprocessing directive

It is used to define symbolic constants

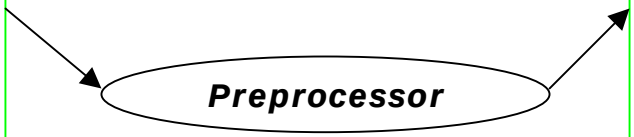
#include is also a preprocessing directive which is used to include contents of a file.

Preprocessing

```
#include<stdio.h>
#define SIZE 54
main( )
{
    int marks[SIZE];

    /* to find average */
    for( j = 0; j < SIZE; j++ ){
    ……;
    }

   ……;
}
```




```
/* contents of stdio.h are kept
    here */
main( )
{
    int marks[54];

    /* to find average */
    for( j = 0; j < 54; j++ ){
    ……;
    }

   ……;
}
```

- **Simply SIZE is replaced by 54 in the source file.**
- **Preprocessing is done first and then compilation is done.**

Preprocessing

- Remember that it is an error to end *#include* or *#define* with a semicolon. These are not C statements.
- A symbolic constant like SIZE is not a variable.
- Advise: Use only UPPERCASE LETTERS for symbolic constants. This distinguishes them from other identifiers.
- There are many more preprocessing directives which are often called as macros, but discussion of these are as usual postponed.

Passing arrays to functions

- *int count[5];*
- *count* is the name of the array and value of *count* is a constant which is the starting address of the array.
- When we say *count[3] = 33;*

  what happens is 33 is stored in the memory location whose address is :

  (count + 3 sizeof (int))*
- So, when we pass count to a function, then we are actually passing an address.
- So, the function which uses this address can in fact modify the original array !

Passing arrays to functions

- Function prototype which takes an array as argument:
  - *return-type function(array-type array-name[ ]);*
  - Optionally array-name can be omitted.
  - Eg: *float find_average(float marks[ ]);*
  - Or, *float find_average(float [ ]);*
- Function definition
  - float find_average(float marks[ ])

    {

    .....;

    }

Functions with arrays

```
void f_1(int [ ]);
main( )
{
  int a[10];
  … …;
 f_1(a);
  … …;
 }
  void f_1(int  b[ ])
  {
    b[4] = 20;/* in fact a[4] in main is assigned with 20 */
    .. …;
  }
```

See the difference

```
void swap(int, int);
main( )
{
   int a[2] = {10, 20};
   swap(a[0], a[1]);
   printf("%d   %d", a[0], a[1]);
 }
   void swap(int b0, int b1)
   {
         int t;
         t = b0, b0 = b1, b1 = t;
         return;
   }
```

```
void swap(int [ ]);
main( )
{
   int a[2] = {10, 20};
   swap(a);
   printf("%d   %d", a[0], a[1]);
 }
   void swap(int b[ ])
   {
         int t;
         t = b[0], b[0] = b[1],
                  b[1] = t;
         return;
   }
```

Multi-dimensional arrays

- Until now what we saw are one dimensional arrays.
  - An element is indexed with a single subscript.
  - A list of elements can be stored.
- To store a table of elements (for example, a matrix) we require a two dimensional array.

2 dimensional array

- Declaration
  - int mat[2][4];

Number of rows

Number of columns

mat[0][0]	mat[0][1]	mat[0][2]	mat[0][3]
mat[1][0]	mat[1][1]	mat[1][2]	mat[1][3]

2 dimensional arrays: initialization

- int mat[2][4] = { {1,2,3,4} , {5,6,7,8} };
- mat[0][0] = 1, mat[0][1] = 2, …, mat[1][3] = 8

- int a[2][2] = { {1}, {3,4} };
- a[0][0] = 1, a[0][1] = 0, a[1][0] = 3, a[1][1] = 4

- int b[2][2] = { 1,3,4};
- b[0][0] = 1, b[0][1] = 3, b[1][0] = 4, b[1][1] = 0

2 dimensional arrays: initialization

- In one dimensional arrays we said that

  int a[ ] = {1,2,3}; /* same as int a[3] = {1,2,3}; */
- Can we do for 2 dim arrays also like that

  int b[ ][ ] = {1,2,3,4,5,6};
- b could be of 2 rows and 3 columns, or of 3 rows and 2 columns.
- To avoid this ambiguity, the column size (second subscript) must be specified explicitly in the declaration.
- int b[ ][2] = {1,2,3,4,5,6}; /*this is OK*/
- b has two columns and three rows

2 dimensional arrays: initialization

- Check the following declaration
- int a[ ][2] = {1,2,3};

- a has 2 rows and 2 columns.
- a[1][1] = 0

- But declaring, int a[2][ ] = {1,2,3};
  will not work !

2 dimensional arrays: addresses

This is the important number which is collected from the declaration

- int a[4][5];
- When you say a[2][3], the location accessed is at address (a + 2*(5*sizeof(int)) + 3*sizeof(int))
- Let a = 100 and sizeof(int) = 2 then the addresses of consecutive cells are as:

	0	1	2	3	4
0	100	102	104	106	108
1	110	112	114	116	118
2	120	122	124	126	128
3	130	132	134	136	138

2 dimensional arrays: addresses

- int a[4][5];

- a is the starting address of the 2 dim. Array
- a[k] is the starting address of the k th row

```
a = 100
a[0] = 100
a[1] = 110
a[2] = 120
a[3] = 130
```

100	102	104	106	108
110	112	114	116	118
120	122	124	126	128
130	132	134	136	138

2 dimensional arrays: functions

- Function prototype

  double f_1(char [ ][4]);

- Number of columns in the argument array must be specified. Otherwise address calculations are not possible.

- Function definition

```
double f_1(char names[ ][4])
{
        ... ...;
}
```

2 dimensional arrays: functions

- **Function call**

```
double  f_1(char  [ ][4]);
main( )
{
        char str[10][4];
        … …;
        v =  f_1(str);
}
```

```
double f_1(char names[ ][4])
{
                … …;
}
```

2 dim arrays: Example: Read 10 words from keyboard

- void read_names(char [ ][16]);

```
/*func protorype*/
main( )
{
        char name[40][16];
        read_names(name);
}
void read_names(char str[ ][16])
{
        for( j =0; j<40; j++)
                scanf("%s", str[ j] );
}
```

Multi-dimensional arrays

- This is simply generalization of 2 dimensional arrays.
- For example
  - int a[2][4][9];
  - int b[ ][3][2] = {1,2,3,4,5,6,7};

This can be left blank if initialization is given

  - Similar rules for functions as for 2 dim. arrays