Arrays in C

- An array is a collective name given to a group of similar quantities.
- *These similar elements could be all integers or all floats or all characters etc.*
- Usually, if is an array of characters is called a “string”. (**“hello world”** is a string array of characters…)
- An array in C Programing language can be defined as number of memory locations each of which can store the same data type and which can be reference through the same variable name (**identifier**).

The array example from last lecture:

```
/* ages.c */

#include <stdio.h>

int ArraySum( int array[ ], int size)
{
  int k, sum = 0;
  for (k = 0; k < size; k++)
    sum += array[ k ];
  return sum;
}

int ArrayAvg( int array[ ], int size)
{
  double sum = ArraySum( array, size );
  return sum / size;
}

int main( )
{
int ages[ 6 ] = {19, 18, 17, 22, 44, 55};
  int avgAge = ArrayAvg( ages, 6 );
  printf(“The average age is %d\n”, ageSum);
  return 0;
}
```

Declaration of an Array

- Arrays must be declared before they can be used in the program. Standard array declaration is as:

```
type variable_name[length of array];
```

- **type**: type specifies the variable type of the element which is going to be stored in the array.

- **variable name**: Any name of the variable through which we are going to address.

- **length of array**: computer will reserve a contiguous* memory space according to length of array in memory.

  *contiguous so far as the program itself is concerned. If a platform uses memory paging then the storage itself may be segmented but it would be transparent to the program itself.

- Example:

```
double height[10];
```

array type : double
array variable name: height
array length :10

- Other examples:

```
float width[20];
int c[9];
char name[20];
```

Array implementation in C

- A key to understanding the nuances of arrays in C: The array identifier is actually a variable that stores the address of the first element in the array. Typically, the address is deferenced* with a offset (variable or fixed), thereby accessing the various elements of the array for reading or modification.

  * deference means accessing the contents of a pointer… Later we will learn about pointers which are variables used to access memory locations and how pointers, arrays, and strings are closely related in C.

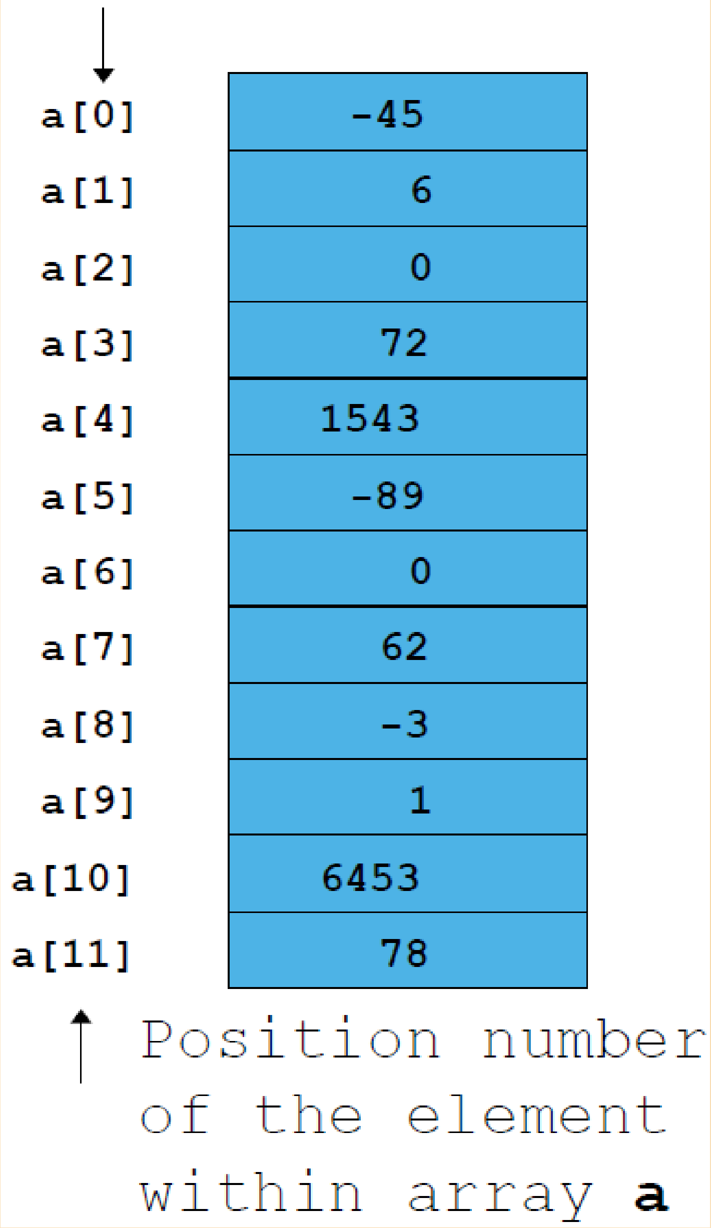
Arrays element location

- So… in C, an array is a group of consecutive memory locations of same name and type.
- How is an individual array element placed in memory?
  e.g : arrayname [position number]

- The first element is at position 0.
- The second element is at position 1.
- The n^th element is at position (n-1).

- Accessing an array variable:
  a[n] //(n+1)th element in array
  a[0],a[1],a[2]……a[n-1]

- Note that all elements of the array at right use the same identifier, “**a**”. There are 12 elements in this array, numbered from 0 to 11:



a[0]	-45
a[1]	6
a[2]	0
a[3]	72
a[4]	1543
a[5]	-89
a[6]	0
a[7]	62
a[8]	-3
a[9]	1
a[10]	6453
a[11]	78

Position number of the element within array **a**

Array Initialization, examples…

- In the following initialization, all three arrays will be of the type: integer:

```
        int a [7],b,c[8],d;
```

- Arrays are NOT auto initialized. Arrays not explicitly initialized at all are left with garbage contents.

- With initialization:

```
        int a[5]={1,2,3,4}; // note that the array size for a is 5
```

- so... if array size > numbers initialized in array then additional trailing values are initialized to Zero

```
        int a[5]={0};       // useful short hand for initializing
                            //   the entire array to zero.
```

Array Initialization, examples…

- If the number of initial values > number of array elements allocated in memory a compiler error is produced:

```
     int a[5]={1,2,3,4,5,6}; // this gives a compiler error
```

- Size is not required if the initializing array is fully declared.

```
     int a[ ]={1,2,3,4,5};    //note that no size is required
                              //between the square brackets.
```

- A mixed set of array and non-array variables can all be declared in the same line if they are of the same type. Here, all variables will be of type integer:

```
     int a[ ]={1,2}, j,k=0;
```

Accessing array elements

- Once accessed with a deference (see slide 3) with an offset using the [] operator, array elements are like normal variables and can be read and modified similarly:

```
c[ 0 ] = 3;
printf( "%d", c[ 0 ] );
```

- You may perform operations and use variables in the subscript. If **x** equals **3** then
  - **c[5 - 2] == c[3] == c[x]**

- Given the following array initialization:

```
int a[ ]={10,20,30,40};
int b,c;
```

  - a[2]=?
  - If b=1,c=2 and we have the following statement in the program:

```
a[b+c]=7;
```

  - The order of precedence of the [ ] operator is low. The index used is 3 (just b+c).
  - a[3] then refers to the 4th element in the array. So 40 will be replaced by new value i.e 7
  - Thus, array ‘a’ now becomes {10,20,30,7}

Char Array

- Character arrays can be initialized using string literals
- String literal literals are specified with double quotes. They are an array of constant characters and are terminated by a null character

```
char string1[] = "first"; // note double quotes, and
                          // no commas between chars
```

- A Null character, which can be explicitly expressed with an escape sequence **\0** in a char or string literal, terminates C-style strings (‘**\0**’ is typically 0)

- Note that `string1` actually has 6 elements

- `string1` is equivalent to the following array declaration:

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

- It is possible to access individual characters using array notation:
  - string1[3] is the character ‘s’

Char Array

- The following gives an error because, with the required terminating (‘\0’), there are actually 6 characters in the string:

```
char string [5]=“hello”;  // this line is the same
                          //   as the next one...
char string [5]={‘h’,’e’,’l’,’l,’l’,’o’,’\0’};
```

- In the same fashion, the line below will compile – the problem is that it will waste some memory with unused string terminations (‘\0’):

```
char string [10]=“hello”; // this line is the same
                          //   as the next one...
char string [10]={‘h’,’e’,’l’,’l,’l’,’o’,’\0’,’\0’,
                                ’\0’,’\0’,’\0’};
```

- Note: string=“hello”; gives a syntax error. We will learn about C-style strings next time.

scanf

- Reads characters until whitespace encountered

```
        char string2[10];
        scanf("%s", string2);
```

- Can write beyond end of array, be careful
- Array name is address of array, so ampersand (**&**) **not** needed for scanf

```
        scanf("%s", &string2);
```

Program with Char Array

```
/* Fig. 6.10: fig06_10.c
 Treating character arrays as strings */
#include <stdio.h>

int main()
{
  char string1[ 20 ], string2[] = "string literal";
  int i;

  printf(" Enter a string: ");
  scanf( "%s", string1 );
  printf( "string1 is: %s\nstring2: is %s\n"
  "string1 with spaces between characters is:\n",
  string1, string2 );

  for ( i = 0; string1[ i ] != '\0'; i++ )
  printf( "%c ", string1[ i ] );

  printf( "\n" );
  return 0;
 }
```

Outline

- 1. Initialize strings
- 2. Print strings
- 2.1 Define loop
- 2.2 Print characters
- individually
- 2.3 Input string
- 3. Print string

Program Output

```
Enter a string: Hello there

string1 is: Hello

string2 is: string literal

string1 with spaces between characters is:

H e l l o
```

Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array without any brackets

```
int myArray[ 24 ];
myFunction( myArray);
```

- Arrays are passed call-by-reference (a copy of the address of the array is passed)
- The name of the array is the address of the first element, so that function knows where the array is stored
- Modifies original memory locations
- Passing array elements
- Passed by call-by-value
- A subscripted name (i.e., `myArray[ 3 ]`) may be passed to a function

Arrays, Functions, and Constants:

- The scope of a variable does not restrict access to the array elements. If we define a function as follows:

```
void ChangeElement(int a[ ], int index, int value){
a[index]=value;
};
```

If size of array is provided in the function prototype or declaration, it is ignored anyway. The square brackets are there only to indicate an array variable is being passed.

- --- and then, somewhere later in the main() function we have the following lines:

```
int a[ ]={1,2};

ChangeElement(a,0,3);  // a becomes{3,2}..even in main

/* unlike non-array variable passing the “array” changes in main
too. Only the address of the array is passed through the
stack(pass-by-copy of address of array) */

printf(“%d\n”,a[0]);  //prints 3
```

Protecting array elements using const

- **const** modifier will *help* protect contents of constant-element arrays by generating some compiler messages. Continuing with same function definition:

```
void ChangeElement(int a[ ], int index, int value){
a[index]=value;
};
```

- --- and then, again, somewhere later in the main() function we have the following lines:

```
.
.
/*...somewhere in main...*/
const int a[ ]={7,8,9};
.
.
ChangeElement (a,1,3); /* at least gives at a compiler
                            warning about the first argument*/
```

Other Functions with const Array

- The following will generate a compiler warning:

```
int AccessElement(int a[ ],int index){
  return(a [index]);
}

/*…somewhere in Main…*/
const int a[7]={0};
int b;
.
.
.
b= AccessElement(a,1);  /*Even though not modifying a in the function,
                          a compiler warning is still generated*/
```

- **Coding rule: Always provide const modifier in parameter types where appropriate even though it is optional :**

```
int AccessElement(const int a[ ],int index){
  return(a [index]);
}

/*…somewhere in Main…*/
const int a[7];
int b;
.
.
.
b= AccessElement(a,1);  // now, no warning is given //
```

Implicit Type Casting

- Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast** (slide 18) operator.

```
/*somewhere in main*/

float f0=0,f1=1,f2=2;
int   i0=0,i1=1,i2=2;
char  c0=0,c1=1,c2=2;
.
.
.
f0=i1/i2;   /*int by int division, the result is cast
                to a float, f0 become 0.0 */
```

Promotion

- So now, if the same lines from the previous page are used, but the change indicated in **RED** occurs:

```
/*somewhere in main*/

float f0=0,f1=1,f2=2;
int   i0=0,i1=1,i2=2;
.
.
.
f0=f1/i2;   /*int by int division, the result is cast
                to a float, f0 become 0.0 */
```

- C only supports dividing by same types. So a temporary copy of i2 as a float is made and used for the operation. This is called **promotion** or **implicit type casting**. f0 is 0.5

- In similar fashion, if we have another line in the code that reads as follows:

```
.
c0=c1/i2;   /*char by int division, the result is cast
                to a character */
```

- This is a character-integer division, char / int, so i1 undergoes promotion to a character.

Demotion

- **Shortening integral types**: Example is assigning an int to char, where bit-truncation occurs. The result is undefined by the language if the value can not be stored in the lower rank type, though typically the result is a truncation at the bit level.

- **Float to int casting** attempts to truncate (remove) fractional part. ***!Not Rounding!***:

```
     int i=1.5; //sets i to 1//
```

- Unsigned to signed casting is particularly dangerous. Hint: think about bit copying

```
     unsigned int i=-1; /*gives a very large
                          positive number!! */
```

Implicit Type Casting through Parameter Passing

- Example:

```
int mult(int a, int b){
  return(a*b);
}


.
.
/*...somewhere in main….*/
.
.
float f0,f1,f2;
.
.
f0=mult(f1,f2); /*parameter passing is like
                   assignments, implicit casting
                   can occur & cause warnings*/

f0=(float)mult((int)f1,int(f2));  /*better to use
                           explicit type casting*/
```