Comp 2401

Arrays and Pointers

Class Notes

2013

The Group of Three

Introduction To Arrays:

In C programming, one of the frequently problem is to handle similar types of data. For example: if the user wants to store marks of 500 students, this can be done by creating 500 variables individually but, this is rather tedious and impracticable. These types of problem can be handled in C programming using arrays.

An array in C Programing can be defined as number of memory locations, each of which can store the same data type and which can be references through the same variable name. It is a collective name given to a group of similar quantities. These similar quantities could be marks of 500 students, number of chairs in university, salaries of 300 employees or ages of 250 students. Thus we can say array is a sequence of data item of homogeneous values (same type). These values could be all integers, floats or characters etc.

We have two types of arrays:

1. One-dimensional arrays.
2. Multidimensional arrays.

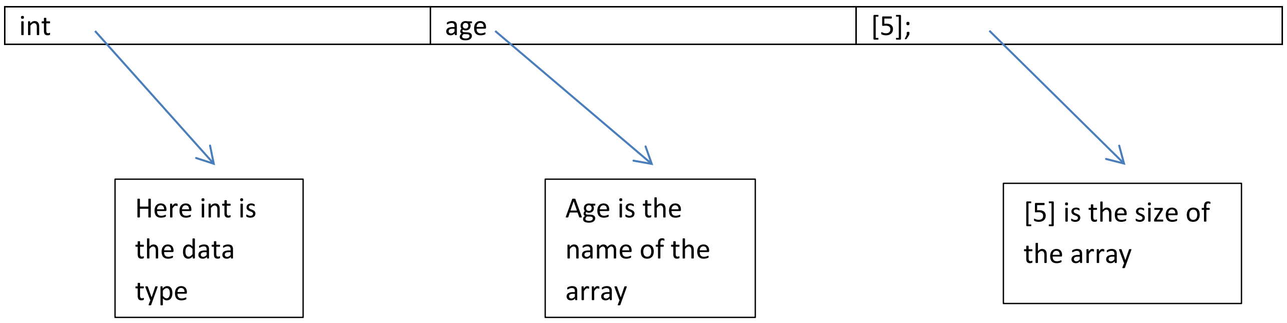
One Dimensional Arrays:

A **one-dimensional array** is a structured collection of components (often called *array elements*) that can be accessed individually by specifying the position of a component with a single *index* value. Arrays must be declared before they can be used in the program. Here is the declaration syntax of one dimensional array:

```
data_type array_name[array_size];
```

Here “data_type” is the type of the array we want to define, “array_name” is the name given to the array and “array_size” is the size of the array that we want to assign to the array. The array size is always mentioned inside the “[]”.

For example:
Int age[5];


int
age
[5];
Here int is the data type
Age is the name of the array
[5] is the size of the array

The following will be the result of the above declarations:

age[0]	age[1]	age[2]	age[3]	age[4]

Initializing Arrays

Initializing of array is very simple in c programming. The initializing values are enclosed within the curly braces in the declaration and placed following an equal sign after the array name. Here is an example which declares and initializes an array of five elements of type int. Array can also be initialized after declaration. Look at the following code, which demonstrate the declaration and initialization of an array.

```
int age[5]={2,3,4,5,6};
```

It is not necessary to define the size of arrays during initialization e.g.

```
int age[]={2,3,4,5,6};
```

In this case, the compiler determines the size of array by calculating the number of elements of an array.

age[0]	age[1]	age[2]	age[3]	age[4]
2	3	4	5	6

Accessing array elements

In C programming, arrays can be accessed and treated like variables in C.

For example:

- scanf("%d",&age[2]);

//statement to insert value in the third element of array age[]

- printf("%d",age[2]);

//statement to print third element of an array.

Arrays can be accessed and updated using its index.An array of n elements, has indices ranging from 0 to n-1. An element can be updated simply by assigning

A[i] = x;
A great care must be taken in dealing with arrays. Unlike in Java, where array index out of bounds exception is thrown when indices go out of the 0..n-1 range, C arrays may not display any warnings if out of bounds indices are accessed. Instead,compiler may access the elements out of bounds, thus leading to critical run time errors.

Example of array in C programming

```
/* C program to find the sum marks of n students using arrays */

#include <stdio.h>

int main(){

        int i,n;
        int marks[n];
        int sum=0;

        printf("Enter number of students: ");
        scanf("%d",&n);

        for(i=0;i<n;i++){
                printf("Enter marks of student%d: ",i+1);
                scanf("%d",&marks[i]); //saving the marks in array
                sum+=marks[i];
        }

        printf("Sum of marks = %d",sum);

return 0;

}
```

Output :

```
Enter number of students: (input by user)3
Enter marks of student1:  (input by user) 10
Enter marks of student2:  (input by user) 29
Enter marks of student3:  (input by user) 11

Sum of marks = 50
```

Important thing to remember in C arrays

Suppose, you declared the array of 10 students. For example: students[10]. You can use array members from student[0] to student[9]. But, what if you want to use element student[10], student[100] etc. In this case the compiler may not show error using these elements but, may cause fatal error during program execution.

Multidimensional Arrays:

C programming language allows the user to create arrays of arrays known as multidimensional arrays. To access a particular element from the array we have to use two subscripts one for row number and other for column number. The notation is of the form array [i] [j] where i stands for row subscripts and j stands for column subscripts. The array holds i*j elements. Suppose there is a multidimensional array array[i][j][k][m]. Then this array can hold i*j*k*m numbers of data. In the same way, the array of any dimension can be initialized in C programming. For example :

```
int smarks[3][4];
```

Here, smarks is an array of two dimension, which is an example of multidimensional array. This array has 3 rows and 4columns. For better understanding of multidimensional arrays, array elements of above example can be as below:

	Col 1	**Col 2**	**Col 3**	**Col 4**
Row 1	smakrs[0][0]	smarks[0][1]	smarks[0][2]	smarks[0][3]
Row 2	smarks[1][0]	smarks[1][1]	smarks[1][2]	smarks[1][3]
Row 3	smarks[2][0]	smarks[2][1]	smarks[2][2]	smarks[2][3]

Make sure that you remember to put each subscript in its own, correct pair of brackets. All three examples below are wrong.

```
        int a2[5, 7];               /* XXX WRONG */

        a2[i, j] = 0;               /* XXX WRONG */

        a2[j][i] = 0;               /* XXX WRONG */
```

would do anything remotely like what you wanted

Initialization of Multidimensional Arrays

In C, multidimensional arrays can be initialized in different number of ways.

```
int smarks[2][3]={{1,2,3}, {-1,-2,-3}};
        OR
int smarks[][3]={{1,2,3}, {-1,-2,-3}};
        OR
int smarks[2][3]={1,2,3,-1,-2,-3};
```

Coding example of Multidimensional Array:

This program asks user to enter the size of the matrix (rows and column) then, it asks the user to enter the elements of two matrices and finally it adds two matrix and displays the result.

Source Code to Add Two Matrix in C programming

```
#include <stdio.h>

int main(){
        int r,c;
        int a[r][c];
        int b[r][c];
        int sum[r][c;

        printf("Enter number of rows (between 1 and 100): ");
        scanf("%d",&r);

        printf("Enter number of columns (between 1 and 100): ");
        scanf("%d",&c);

        printf("\nEnter elements of 1st matrix:\n");

/* Storing elements of first matrix entered by user. */

        for(int i=0;i<r;++i){
                for(int j=0;j<c;++j){
                printf("Enter element a%d%d: ",i+1,j+1);
                scanf("%d",&a[i][j]);
                }
        }

/* Storing elements of second matrix entered by user. */

        printf("Enter elements of 2nd matrix:\n");
        for(int i=0;i<r;++i){
                for(int j=0;j<c;++j){
                printf("Enter element a%d%d: ",i+1,j+1);
                scanf("%d",&b[i][j]);
                }
        }

/*Adding Two matrices */

        for(int i=0;i<r;++i)
                for(int j=0;j<c;++j)
                sum[i][j]=a[i][j]+b[i][j];

/* Displaying the resultant sum matrix. */
```

```
          printf("\nSum of two matrix is: \n\n");
          for(int i=0;i<r;++i){
                    for(int j=0;j<c;++j){
                    printf("%d   ",sum[i][j]);
                              if(j==c-1)
                              printf("\n\n");
                    }

          }

return 0;

}
```

Program Output:

```
Enter number of rows (between 1 and 100): 3
Enter number of rows (between 1 and 100): 2

Enter elements of 1st matrix:
Enter element a11: 4
Enter element a12: -4
Enter element a21: 8
Enter element a22: 5
Enter element a31: 1
Enter element a32: 0
Enter elements of 2nd matrix:
Enter element a11: 4
Enter element a12: -7
Enter element a21: 9
Enter element a22: 1
Enter element a31: 4
Enter element a32: 5

Sum of two matrix is:

8     -11

17     6

5      5
```

Dynamic Arrays and Resizing

Arrays by definition are static structures, meaning that size cannot be changed during run time. When an array is defined as

```
int A[n];
```

then A is considered a static array and memory is allocated from the run time stack for A. When A goes out of scope, the memory is deallocated and A no longer can be referenced. C allows dynamic declaration of an array as follows:

```
int* A = (int*)malloc(n* sizeof(int))
```

The above code declares a memory block of size n*sizeof(int) that can be accessed using the pointer A. For example, A can be initialized as follows:

```
int i;
for (i=0; i<n; i++)
A[i] = 0;
```

Note that although A was declared as a pointer, A can be treated as an array. The difference between

int A[10] and int* A = malloc(10*sizeof(int)) is that latter is assigned memory in the dynamic heap (and

hence must be managed by the programmer) and former is assigned memory from the run time stack (and hence managed by the compiler). Static arrays are used when we know the amount of bytes in array at compile time while the dynamic array is used where we come to know about the size on run time. Arrays can also be initialized using the calloc() functions instead of the the malloc(). Calloc function is used to reserve space for dynamic arrays. Has the following form.

```
array =(cast-type*)calloc(n,element-size);
```

Number of elements in the first argument specifies the size in bytes of one element to the second argument. A successful partitioning, that address is returned, NULL is returned on failure. For example, an int array of 10 elements can be allocated as follows.

int * array = (int *) calloc (10, sizeof (int));

Note that this function can also malloc, written as follows.

int * array = (int *) malloc (sizeof (int) * 10)

Arrays and Functions

In C, Arrays can be passed to functions using the array name. Array name is a const pointer to the array. both one-dimensional and multi-dimensional array can be passed to function as argument. Individual element is passed to function using pass by value. Original Array elements remain unchanged, as the actual element is never passed to function. Thus function body cannot modify original value in this case. If we have declared an array ‘array [5]‘ then instead of passing individual elements, a for loop is useful in this case to pass all 5 elements of the array.

Passing One-dimensional Array In Function

C program to pass a single element of an array to function

```
#include <stdio.h>
void display(int a)
  {
  printf("%d",a);
  }
int main(){
  int c[]={2,3,4};
  display(c[2]);  //Passing array element c[2] only.
  return 0;
}
```

Output

```
4
```

Single element of an array can be passed in similar manner as passing variable to a function.

Passing entire one-dimensional array to a function

While passing arrays to the argument, the name of the array is passed as an argument(,i.e, starting address of memory area is passed as argument).

C program to pass an array containing age of person to a function. This function will return average age and display the average age in main function.

```
#include <stdio.h>
float average(float a[]);
int main(){
    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
    avg=average(c);   /* Only name of array is passed as argument. */
    printf("Average age=%.2f",avg);
    return 0;
  }
float average(float a[]){
    int i;
    float avg, sum=0.0;
    for(i=0;i<6;++i){
      sum+=a[i];
    }
    avg =(sum/6);
    return avg;
}
```

Output

```
Average age=27.08
```

Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example to pass two-dimensional arrays to function

```
#include
void Function(int c[2][2]);
int main(){
   int c[2][2],i,j;
   printf("Enter 4 numbers:\n");
   for(i=0;i<2;++i)
      for(j=0;j<2;++j){
         scanf("%d",&c[i][j]);
     }
   Function(c);   /* passing multi-dimensional array to function */
   return 0;
}
void Function(int c[2][2]){
/* Instead to above line, void Function(int c[][2]){ is also valid */
   int i,j;
   printf("Displaying:\n");
   for(i=0;i<2;++i)
     for(j=0;j<2;++j)
        printf("%d\n",c[i][j]);
```

Output

```
Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5
```

Introduction to Pointers

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PCs integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. We have integers, long integers and short integers which you can read up on in any basic text on C. This document assumes the use of a 32 bit system with 4 byte integers.

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name k by writing:

```
int k;
```

On seeing the "int" part of this statement the compiler sets aside 4 bytes of memory to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol k and the relative address in memory where those 4 bytes were set aside.

Thus, later if we write:

```
 k = 2;
```

We expect that, at run time when this statement is executed, the value 2 will be placed in that memory location reserved for the storage of the value of k. In C we refer to a variable such as the integer k as an "object".

In a sense there are two "values" associated with the object k. One is the value of the integer stored there and the other the "value" of the memory location, i.e., the address of k. Some texts refer to these two values with the nomenclature rvalue.

Now consider the following:

```
int j, k;
 k = 2;
 j = 7;    <-- line 1
 k = j;    <-- line 2
```

Here the compiler interprets the j in line 1 as the address of the variable j and creates code to copy the value 7 to that address. In line 2, however, the j is interpreted as its rvalue. That is, here the j refers to the value stored at the memory location set aside for j. So,7 is copied to the address designated to k.

In C when we define a pointer variable we do so by preceding its name with an asterisk. We also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int *ptr;
```

ptr is the name of our variable, the '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The int says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. A pointer initialized in this manner is called a "null" pointer.

The actual bit pattern used for a null pointer may or may not evaluate to zero since there is no value assigned to it. Thus, setting the value of a pointer using the NULL macro, as with an assignment statement such as ptr = NULL, guarantees that the pointer has become a null pointer.

Suppose now that we want to store in ptr the address of our integer variable k. To do this we use the unary & operator and write:

```
ptr = &k;
```

The "dereferencing operator" is the asterisk and it is used as follows:

```
*ptr = 7;
```

will copy 7 to the address pointed to by ptr. Thus if ptr "points to" (contains the address of) k, the above statement will set the value of k to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself.

Similarly, we could write:

```
printf("%d\n",*ptr);
```

to print to the screen the integer value stored at the address pointed to by ptr.

Arrays and Pointers

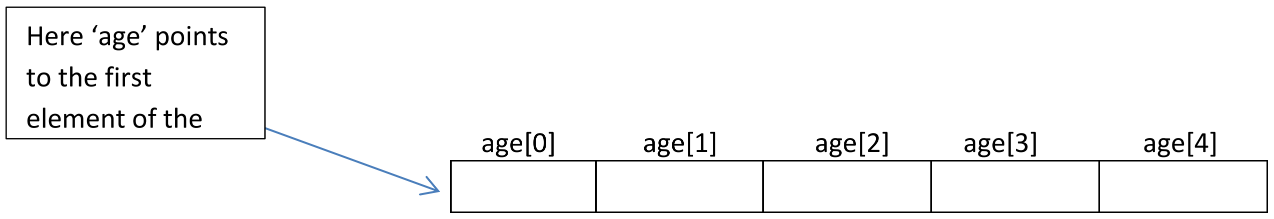
Arrays and pointers are closely related in C. In fact an array declared as

```
int A[10];
```

can be accessed using its pointer representation. The name of the array A is a constant pointer to the first element of the array. So A can be considered a const int*. Since A is a constant pointer, A = NULL would be an illegal statement. Arrays and pointers are synonymous in terms of how they use to access memory. But, the important difference between them is that, a pointer variable can take different addresses as value whereas, in case of array it is fixed.

Consider the following array:

```
Int age[5];
```



In C , name of the array always points to the first element of an array. Here, address of first element of an array is &age[0]. Also, age represents the address of the pointer where it is pointing. Hence, &age[0] is equivalent to age. Note, value inside the address &age[0] and address age are equal. Value in address &age[0] is age[0] and value in address age is *age. Hence, age[0] is equivalent to *age.

C arrays can be of any type. We define array of ints, chars, doubles etc. We can also define an array of pointers as follows. Here is the code to define an array of n char pointers or an array of strings.

```
char* A[n];
```

each cell in the array A[i] is a char* and so it can point to a character. Now if you would like to assign a string to each A[i] you can do something like this.

```
A[i] = malloc(length_of_string + 1);
```

Again this only allocates memory for a string and you still need to copy the characters into this string. So if you are building a dynamic dictionary (n words) you need to allocate memory for n char*'s and then allocate just the right amount of memory for each string.

In C, you can declare an array and can use pointer to alter the data of an array. This program declares the array of six element and the elements of that array are accessed using pointer, and returns the sum.

```
//Program to find the sum of six numbers with arrays and pointers.
#include <stdio.h>
int main(){
  int i,class[6],sum=0;
  printf("Enter 6 numbers:\n");
  for(i=0;i<6;++i){
      scanf("%d",(class+i)); // (class+i) is equivalent to &class[i]
      sum += *(class+i); // *(class+i) is equivalent to class[i]
  }
  printf("Sum=%d",sum);
  return 0;
}
```

Output

```
Enter 6 numbers:
2
3
4
5
3
4
Sum=21
```

Pointer	Array
A pointer is a place in memory that keeps address of another place inside	**An array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location.**
Allows us to indirectly access variables. In other words, we can talk about its address rather than its value	**Expression a[4] refers to the 5th element of the array a.**
Pointer can't be initialized at definition	**Array can be initialized at definition. Example** **int num[] = { 2, 4, 5}**
Pointer is dynamic in nature. The memory allocation can be resized or freed later.	**They are static in nature. Once memory is allocated , it cannot be resized or freed dynamically**

Starting to think like a C programmer

Now that we have spent some time studying, using talking about C language. You may have been trying to think like a Java programmer and convert that thought to C. Now it is time to think like a C programmer. Being able to think directly in C will make you a better C programmer.

Here are 15 things to remember when you start a C program from scratch.

1. include <stdio.h> and all other related headers in all your program.
2. Declare functions and variables before using them
3. Better to increment and decrement with ++ and -- operators.
4. Better to use x += 5 instead of x = x +5
5. A string is an array of characters ending with a '\0". Don't ever forget the null character
6. Array of size n has indices from 0 to n-1. Although C will allow you to access A[n] it is very dangerous
7. A character can be represented by an integer (ASCII value) and can be used as such
8. The unary operator & produces an address
9. The unary operator * dereference a pointer
10. Arguments to functions are always passed by value. But the argument can be an address of just a value
11. For efficiency, pointers can be passed to or return from a function
12. Logical false is zero and anything else is true
13. You can do things like for(;;) or while(i++) for program efficiency and understanding
14. Use /* .. */ instead of //, it makes the code look better and readable
15. The last and most important one, always compile your program before submitting or showing to someone. Don't assume that your code is compliable and contains no errors. Try using –std=c99, which is c99 standard. Its better. (Although, c11 is also on its way but not a standard at the moment for all machines)

Bibliography

- "The C Programming Language" 2nd Edition
  B. Kernighan and D. Ritchie
  Prentice Hall
  ISBN 0-13-110362-8 **document.**

- "Algorithms in C"
  Robert Sedgewick
  Addison-Wesley
  ISBN 0-201-51425-7

- http://www.programiz.com

- http://www.lysator.liu.se/c/bwk-tutor.html#array

- http://www.eskimo.com/~scs/cclass/int/sx9.html