Arrays and Pointers: One-dimensional arrays, Pointers - I

An array in C is a collection of elements of the same data type stored in contiguous memory locations and accessed using a single name and an index. Arrays are used to store multiple values efficiently, such as a list of student marks or employee names, instead of declaring individual variables for each value. A pointer in C is a variable that stores the memory address of another variable. Pointers allow indirect access and manipulation of data stored in different memory locations, a powerful and essential feature of C for tasks such as efficient memory management and the construction of complex data structures. In C, arrays and pointers are closely related concepts. An array name by itself is treated as a constant pointer, and pointers, like arrays, can be indexed.

One-dimensional array: We use homogeneous data in many problems. For example, suppose a class has 5 students, and we may want to calculate the average mark. For this, we declare `int mark1,mark2,mark3,mark4,mark5;`
and read/assign the individual marks. Note that we need to write separate statements for each mark when reading from the terminal/file. Then, we calculate the average marks. But suppose the class has 200 students. Then, declaring individual identifier for each student is not feasible. Also, reading each identifier from the terminal/file is not feasible. Since students' marks are integers, we are dealing with homogeneous data. An array becomes useful for this kind of situation. For example, in the case of 5 students, we declare
`int mark[5];`
where `mark[0]` corresponds to marks of first student, `mark[1]` for second student, `mark[2]` for third student, `mark[3]` for fourth student, and `mark[4]` for fifth student. *Note that the index runs from 0 to 4.* For example, consider the following code, which implements the above.

```
#include<stdio.h>
int main()
{
 int i,mark[5],sum;
 double avg;

 for(i=0;i<5;i++)    // read marks
 {
         printf("Enter mark for student %d :",i+1);
         scanf("%d",&mark[i]);
 }
 sum=0;
 for(i=0;i<5;i++)
 {
  sum +=mark[i];
```

```
 }
 avg=sum/5.0;
 printf("Average=%.2f\n",avg);
 return 0;
}
```

Listing 43: C program "cavmrk.c"

Below is a sample input-output:

```
Enter mark for student 1 :45
Enter mark for student 2 :67
Enter mark for student 3 :79
Enter mark for student 4 :83
Enter mark for student 5 :38
Average=62.40
```

Similarly, if we declare
`#define N 100`
`double a[N];`
then index runs from `0` to `99`. Note that the array dimension is defined using a symbolic constant `N`. The advantage is that if we want to work with dimension `400`, we just need to change `100` to `400` in the `#define` directive.

If the number of marks is large, it is infeasible to read these marks from the keyboard (as done in the above program). Usually, we then read these marks from a data file. Reading from a data file will be discussed later. When we write a program using arrays, the dimension is usually taken to be the maximum possible value. For example, we know that the class strength is 70 this year, but it may be 80 next year. For this, we write a code using `100` for `N`. In the code, we will read the actual number of students in the class. The following code calculates the number of students who got more than average marks. To assign marks, we use the random number generator function `rand()` that returns an integer between `0` and `RAND_MAX` (both inclusive). The value of `RAND_MAX` is usually found `stdlib.h`. In my Linux system it is defined as
`#define RAND_MAX 2147483647`
Assuming the maximum mark to be `100`, we execute `rand()%101` to get a random mark between `0` and `100` (both inclusive). We also print the assigned marks with 10 marks in a line.

```
#include<stdio.h>
#include<stdlib.h>
#define N 100
int main()
{
```

```
 int i,mark[N],sum,count,n;      // n actual student number
 double avg;
 printf("Enter actual no. of students: ");
 scanf("%d",&n);
 for(i=0;i<n;i++)    // assign marks
 {
  mark[i]=rand()%101;
 }
 printf("Marks of the students are:\n");
 for(i=0;i<n;i++)         // printing 10 marks in one line
 {
  printf("%4d",mark[i]);
  if((i+1)%10==0)
  {
   printf("\n");
  }
 }
 printf("\n");
 sum=0;
 for(i=0;i<n;i++)
 {
  sum +=mark[i];
 }
 avg=sum/(double) n;
 count=0;
 for(i=0;i<n;i++)   // cal. no. of students > avg
 {
  if(mark[i]>avg)
  {
   count++;
  }
 }
 printf("Average marks=%.2f\n",avg);
 printf("No. of students with marks > avg is %d\n",count);
 return 0;
}
```

Listing 44: C program "nogavg.c"

Below is a sample input-output.

```
Enter actual no. of students: 57
Marks of the students are:
  41  65  31  41  19  15  72  11  78  69
  37  23  29  63  75   4   5  49  75  99
  27  61  62  17  79  61  22  13  49  71
  61   8  81  67  80  47  83  88  30  12
  74  78  33  23  58  83  76  59  77  85
  25  18  98  16 100  79  88
Average marks=52.46
No. of students with marks > avg is 31
```

One-dimensional array initialization : The assignment for a small one-dimensional array can be done using initialization in the declaration statement. For example, the purpose of the following code segment

```
double a[5];
a[0]=2.2;
a[1]=-3.0;
a[2]=1.1;
a[3]=7.0;
a[4]=-2.0;
```

can be achieved by a single line statement

```
double a[5]={2.2,-3.0,1.1,7.0,-2.0};
```

Here, `2.2,-3.0,1.1,7.0,-2.0` are the initializers. When a list of initializers is shorter than the number of array elements to be initialized, the remaining elements are initialized to zero. For example,
`double a[5]={2.2,-3.0,1.1}` is equivalent to `double a[5]={2.2,-3.0,1.1,0.0,0.0}`
Due to this, we can initialize any small or large one-dimensional arrays to `0`. For example,
`int a[1000]={0}` assigns `0` to each of `a[0],a[1],.........,a[999]`.
If we omit the array dimension but use a list of initializers, the array dimension will be determined by the number of initializers. For example,
`double a[]={2.1,3.0,4.2}` is equivalent to `double a[3]={2.1,3.0,4.2}`
In such cases, a character array behaves a little differently from other arrays. The case character array will be discussed later.

Pointers:

Simple variables such as `int a` or `double b` in a program are stored in a fixed number of bytes at specific memory locations, or addresses, in the machine. For example, an `int` variable usually uses 4 bytes and a `double` variable uses 8 bytes of memory.

A pointer is a variable in C that is used to hold the memory address of a variable. Since a pointer is itself a variable, the address of a pointer can be stored in another pointer. If `a` is a variable, then `&a` is the location, or address, in memory of its stored value. The address operator `&` is unary and has the same precedence and right-to-left associativity as the other unary operators. Pointer variables can be declared in programs and then used to take addresses as values. The expression `int *p` declares `p` to be of type pointer to `int`. Thus, we can assign the address of `int` variable to `p`. The indirection or dereferencing operator `*` is unary and has the same precedence and right-to-left associativity as the other unary operators. If `p` is a pointer, then `*p` is the value of the variable for which `p` is the address. Note that the direct value `p` is a memory address and `*p` is the content of `p`, i.e., the value stored in the address. Suppose we declare `int *p` and assign some address of `int` variable to `p`. Assume that an `int` variable takes 4 bytes of memory. Then `*p` is the value of `int` stored in 4 bytes of memory whose starting address is `p`. Consider the following code in which `a` is an `int` variable and `b` is a double variable. Also, `p` is a pointer variable which can hold the address of an `int` variable, and `r` is a pointer variable which can hold the address of a `double` variable. Further, `q` is a pointer to a pointer, a variable that stores the memory address of another pointer (which in turn points to an integer). We first print the size of different variables. For this, we use the `sizeof()` operator, which returns an `unsigned long int`, and we print it using the `%lu` format. As can be seen from the output, in my MacOSX, an `int` takes 4 bytes and a `double` takes 8 bytes. On the other hand, all the pointer variables (which store addresses) take 8 bytes. Next we print the addresses of the variables `a,b,p,q,r`. Note that these addresses are not fixed. If you run the program, you (usually) get different addresses in each run. Since `p` is a pointer for an `int` variable, we can assign address of `a` to `p` by `p=&a`. Similarly, we have `r=&b`. Since the pointer `q` can hold address of a pointer (which holds address of `int` varibale), we write `q=&p`.

```
#include<stdio.h>
int main()
{
 int a=2,*p,**q;
 double b=4.0,*r;
 p=&a;
 r=&b;
 q=&p;
 printf("Size of int: %lu bytes\n",sizeof(a));
 printf("Size of double: %lu bytes\n",sizeof(b));
 printf("Size of int *: %lu bytes\n",sizeof(p));
 printf("Size of double *: %lu bytes\n",sizeof(r));
 printf("Size of int **: %lu bytes\n",sizeof(q));
```

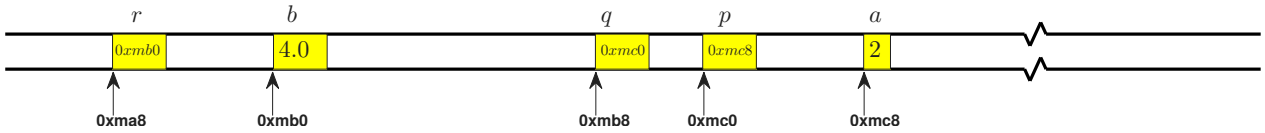

Figure 1: Memory arrangement for Listing 43. Here **0xm** stands for **0x16d2331**

```
 printf("&a=%p  %p   %p\n",&a,p,*q);
 printf("&b=%p  %p\n",&b,r);
 printf("&p=%p  %p\n",&p,q);
 printf("&q=%p\n",&q);
 printf("&r=%p\n",&r);
 printf("a=%d   %d  %d\n",a,*p,**q);
 printf("b=%.2f  %.2f\n",b,*r);
 return 0;
}
```

Listing 45: C Program "adsimp.c"

```
Size of int: 4 bytes
Size of double: 8 bytes
Size of int *: 8 bytes
Size of double *: 8 bytes
Size of int **: 8 bytes
&a=0x16d2331c8  0x16d2331c8   0x16d2331c8
&b=0x16d2331b0  0x16d2331b0
&p=0x16d2331c0  0x16d2331c0
&q=0x16d2331b8
&r=0x16d2331a8
a=2   2  2
b=4.00  4.00
```

The variables `a,b,p,q,r` are stored in specific addresses of the memory as shown in Figure 1. To print an address we use `%p` format which prints the address in hexadecimal address. We have initialized `a` to 2 and `b` to 4.0. We have also assigned `p=&a`, `q=&p` and `r=&b` as discussed above. Thus, the value of `p` is address of `a` (see Figure 1). Consequently, the value of `*p` is the value of `a`. Similar is the case for `b` and `r`. Since `q` is address of `p`, we have `*q` equals `p` and `**q` equals `*p` equals `a`. These are illustrated in Figure 1 and are confirmed by the output.

Since address is a number, we can also do arithmetic with pointers. Note that a pointer can be initialized during declaration too. For example, consider the following code:

```
#include<stdio.h>
int main()
{
 int i=3,j,*p=&i,*q=&j,*r;
 double *s;
 s=p;   // A warning will be issued
 j=7;
 printf("%d\n",p==&j); // 0 since p=&i
 *p=*p**q/ *q + 7;  // careful about * after /
 printf("%d\n",i); // 10 is printed
 *(r=q)*=*p;
 printf("%d\n",j); // 70 is printed
 return 0;
}
```

Listing 46: C Program "admnp.c"

```
0
10
70
```

In the above code, we have initialized `p` with the address of `i` and `q` with address of `j`. During compilation, a warning will be issued regarding the expression `s=p`. This is because `p` has an address of an `int` variable `i` and we are assigning this to a pointer which ideally should hold the address of a `double` variable. The expression `p==&j` is false since `p=&i`, and thus `0` is printed. Next, we consider the statement `*p=*p**q/ *q + 7;` We first concentrate on the right side. Since the de-reference operator `*` has higher priority than the multiplication operator (using the precedence and associativity), the right side is equivalent to `(((*p)*(*q))/(*q))+7`. Thus, it equals to `(((3)*(7))/(7))+7=10`. Now `10` is assigned to `i` since `p` has the address of `i`. Note the gap after `/` in the expression `*p**q/ *q + 7`. If we write `*p**q/*q + 7`, then `/*` is interpreted as the start of a comment, which gives an error. Finally, consider the last statement `*(r=q)*=*p;` Note that the dereference operator `*` and address operator `&` are unary operator are of equal priority. Thus, the above statement is equivalent to `(*(r=q))*=(*p)` which equals `(*(r=q))=(*(r=q))*(*p);`. Now consider the right side `(*(r=q))*(*p)`, where `q` is assigned to `r` and `(r=q)` has value `q`. Thus, `(*(r=q))*(*p)`=`(*q)*(*p)`=`(7)*(10)=70`. Consequently, `70` is assigned to `*q` which is the same as `*r`. Thus, `j` becomes `70`.

Finally, we extend the operator precedence and associativity table discussed in Lecture 12 with the new operators (`[]`, `sizeof()`, `*`, `&`) introduced here. Note that the array subscript operator `[]` has higher precedence than the de-eference operator `*`. Thus, the declaration `int *p[5]` is equivalent to `int *(p[5])`. Thus, `p` is an array of 5 element where each of them

can hold address of an `int` variable. On the other hand, the declaration `int (*p)[5]` means `p` is a pointer which holds address of a 5 element array in which each element is an integer.

Category	Operator	Associativity
Postfix operators	`( )` (parenthesis), `[ ]` (array subscript), `++` (postfix increment), `--` (postfix decrement),	left to right
Unary operators	`+` (unary plus), `-` (unary minus), `++` (prefix increment), `--` (prefix decrement), sizeof, (type) (type casting), `!` (logical not), `*` (de-reference), `&` (address-of)	right to left
Multiplicative operators	`*` (multiplication), `/` (division), `%` (modulus)	left to right
Additive operators	`+` (addition), `-` (subtraction)	left to right
Relational operators	`<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to)	left to right
Equality operators	`==` (equal to), `!=` (not equal to)	left to right
Logical and	`&&`	left to right
Logical or	`||`	left to right
Conditional (ternary) operator	`?:`	right to left
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	right to left

Return to Lecture Topics