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

Note that we were unsuccessful in swapping two `int` values using a function in Lecture 16 (see Listing 40). We now show that the same can be accomplished with pointers using a function. The code is given below. We discuss the variables of `main()` and function `crswap()` separately.

```
#include<stdio.h>
void crswap(int *,int *);      // function prototype
int main()
{
int a=5,b=3;
printf("In main(): &a=%p   &b=%p\n",&a,&b);
printf("Before swap in main(): a=%d  b=%d\n",a,b);
crswap(&a,&b);
printf("After swap in main(): a=%d  b=%d\n",a,b);
return 0;
}
void crswap(int *a,int *b)
{
 int t;
 printf("In crswap(): &a=%p   &b=%p   &t=%p\n",&a,&b,&t);
 printf("In crswap(): a=%p    b=%p\n",a,b);
 t=*a;
 *a=*b;
 *b=t;
}
```

Listing 47: C Program "vswap.c"

```
In main(): &a=0x16cfe71f8   &b=0x16cfe71f4
Before swap in main(): a=5  b=3
In crswap(): &a=0x16cfe71b8   &b=0x16cfe71b0   &t=0x16cfe71ac
In crswap(): a=0x16cfe71f8    b=0x16cfe71f4
After swap in main(): a=3  b=5
```

main():

We initialize `int` variables `a=5` and `b=3` in the main. We also print the addresses of `a` and `b` in the main. Next, we call the `void` function `crswap()` with arguments `&a` and `&b`, that is, the function call is `crswap(0x16d3f31f8,0x16d3f31f4)`. Thus, we are passing two address values. Since the arguments are addresses of `int` variables, this is clearly reflected in the function prototype. First panel in Fig. 1 shows the memory configuration before `crswap()` is called.

crswap():

Now `a`, `b`, and `t`, declared in the function definition, are local to the function `crswap()`. I deliberately use `a` and `b` in the function definition to emphasize the difference with `a` and `b` of `main()`. Variables `a` and `b` defined in `main()` are `int` type. However, `a` and `b` defined in `crswap()` are `int *` type, that is, they can hold addresses of variables of type `int`. The `int *` variables `a` and `b` are placeholders for the function call, that is, they receive the values sent from `main()` via `crswap()`. Hence, in the function `crswap()`, `a=0x16d3f31f8` and `b=0x16d3f31f4`, which is verified from the output of the `printf()` inside the function. Note that `a`, `b` and `t` are stored in different addresses, which are also printed inside `crswap()`. Second panel in Fig. 1 shows the memory configuration after the declaration `int t`.

The expression `t=*a` assigns `int` value `5` stored at address `a` to `t`. The expression `*a=*b`

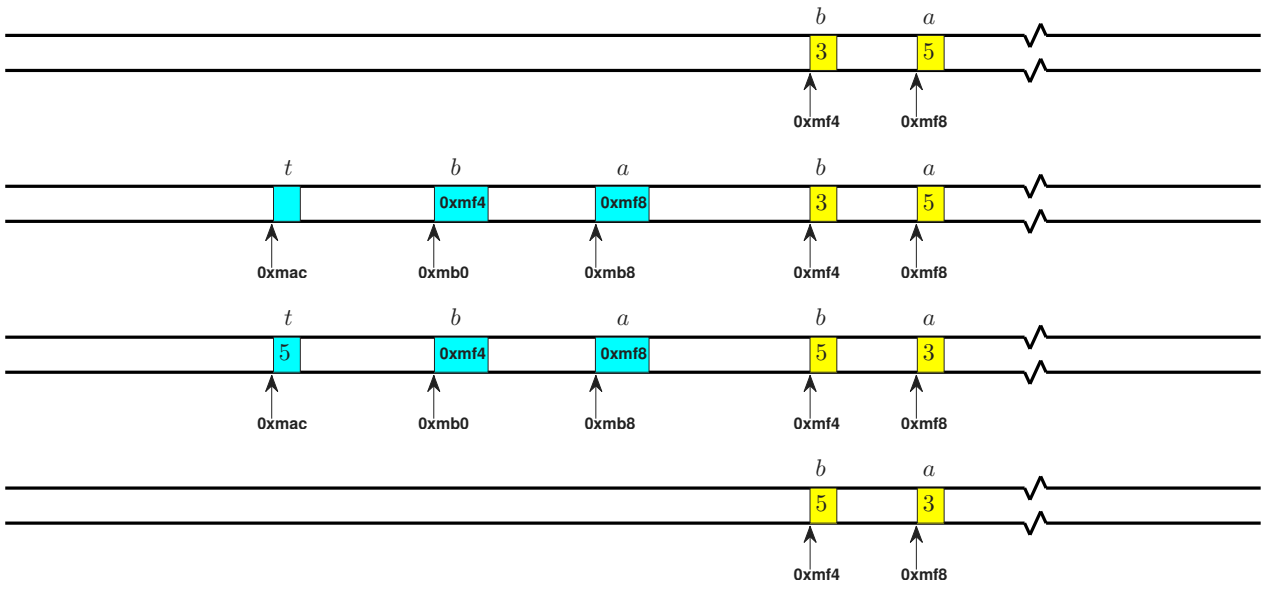

Figure 1: Memory arrangement for Listing 47. Here, **0xm** stands for **0x16cfe71**. Yellow and cyan backgrounds correspond to variables of main() and crswap().

assigns the `int` value `3` stored at address `b` to the address `a`. Finally, the expression `*b=t` causes the value `5` stored at t to be assigned to the address `b`. Third panel in Fig. 1 shows the memory configuration before the function call return to `main()`. Finally, the last panel shows the memory configuration when the function call returns to `main()`. Clearly, the intended swapping has been accomplished.

Relationship between one-dimensional arrays and pointers:

An array name by itself is a pointer which is the address of the first element of the array. Hence, if we declare `int a[5]`, then the name of the array `a` is a (fixed) pointer and `a` is same as `&a[0]`. A pointer variable can be subscripted like an array. Hence, a pointer and an array behave similarly in many situations, but there is a subtle difference as well. Consider the following declaration

```
int a[5],*p,i=5;
```

As discussed before, array name a is equivalent to &a[0]. Note that the array elements are stored contiguously. Now, a+1 is equivalent to &a[0]+1. If an int takes 4 bytes of memory, then a+1 denotes the address that is obtained by adding 4 bytes to &a[0], i.e., a+1=&a[0]+1=&a[1]. In case of double pointer, adding 1 causes advancement by 8 bytes (assuming that a double value takes 8 bytes of memory). Thus, writing a+i is equivalent to &a[i] and conversely *(a+i) is equivalent to content of &a[i], which is a[i]. Thus, *(a+i) is equivalent to a[i]. Now let we write p=a, i.e., p=&a[0]. Now p[i]=*(p+i)=*(a+i)=a[i]. Thus, p behaves like the array a. However, a is an array with 5 contiguous places reserved where each place can hold an integer value. On the other hand, p is pointer variable which can hold address of an int variable. We can make the statement p=a+1 followed by p +=2. Now p[1] is equivalent to a[4]. To see this, the first statement is p=&a[1] and the second statement is p=&a[1]+2 which is equivalent to p=&a[3]. Now

p[1]=*(p+1)=*(&a[3]+1)=*(&a[4])=a[4]

Note that though a is a pointer as well, we CANNOT write a=p or a=a+2. This is because a is a fixed pointer which cannot be changed. Also, note that we can assign p=&i, then p[0]=*(p+0)=*p=*(&i)=i=5. Now p[2]=*(p+2)=*(&i+2). Now &i+2 is the address that is obtained by adding 8 bytes to the address of i. Now the value stored at this address is undefined since this place may not be in use or it may be in use by some other variables (like system variables). Now consider another code shown below.

```
#include<stdio.h>
int main()
{
 int a[5]={2,3,4,9,8},*p,i;
 p=a+2;
 printf("&i=%p   &p=%p\n",&i,&p);
 for(i=0;i<5;i++)
 {
    printf("&a[%d]=%p\n",i,a+i);
  }
 printf("p=%p\n",p);
 return 0;
}
```

Listing 48: C Program "carray.c"

Here is the output:

```
&i=0x16f4a31dc   &p=0x16f4a31e0
&a[0]=0x16f4a31f0
&a[1]=0x16f4a31f4
```

```
&a[2]=0x16f4a31f8
&a[3]=0x16f4a31fc
&a[4]=0x16f4a3200
p=0x16f4a31f8
```

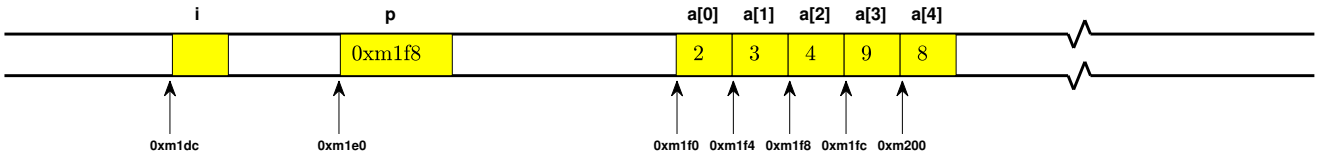

Figure 2: Memory arrangement for Listing 48. Here, **0xm** stands for **0x16f4a3**.

Figure 2 shows the memory configuration for the code given in Listing 48. Clearly, the gap between consecutive array elements addresses is 4 since an `int` take 4 bytes of memory. We have assigned `p=a+2`, which is equivalent to `p=&a[0]+2=&a[2]`, as verified by the output of `p`. We have also printed the addresses of the variables `i` and `p`. Note that `p` takes 8 bytes of memory in my MacOSX.

Since `a[i]=*(a+i)`, we conclude the `i[a]=*(i+a)=*(a+i)=a[i]`. However, we usually write `a[i]` instead of `i[a]`. Since `a+i` is equivalent to `&a[i]`, the expression `scanf("%d",&a[i])` is equivalent to `scanf("%d",a+i)`. If we want to assign `5` to `a[0]`, we write `a[0]=5` or `*a=5`. The same can be accomplished using `p[-2]=5`, since `p[-2]=*(p-2)=*(&a[2]-2)=*(a+0)=a[0]`. Suppose that we want to print the elements of the array. This can be done using the following code:

```
for(i=0;i<5;i++)
{
 printf("a[%d]=%d\n",i,a[i]); // or printf("a[%d]=%d\n",i,p[-2+i]);
}
```

The statement in the comment can also be used to do the same. We can accomplish the same without using the `int` variable `i`.

```
for(p=a;p<a+5;p++)
{
 printf("%d\n",*p);
}
```

Note that now the pointer `p` holds the address of `&a[0]+5`, which is equivalent to `&a[4]+1`. Since the address `&a[4]=0x16f4a3200`, the value of `p` after the execution of the above code will be `0x16f4a3204`.

One-dimensional array as function argument:

Let us consider the program given in Listing 44 in Lecture 17. We now implement the same program with the help of functions. For this, our functions need to access the array elements declared in `main()`. The `main()` function now becomes small.

```
#include<stdio.h>
#include<stdlib.h>
#define N 100
void assign_mrk(int [N],int); // func proto.
void print_mrk(int [],int);   // func proto.
double cal_avg(int *,int);    // func proto.
void gt_avg(int *,int *, double, int);  //func proto.
int main()
{
 int mark[N],count,n;      // n actual student number
 double avg;
 printf("Enter actual no. of students: ");
 scanf("%d",&n);
 assign_mrk(mark,n);      //function to assign marks
 print_mrk(mark,n);       // function to print marks
 avg=cal_avg(mark,n);     // function to find avg mark
 gt_avg(mark,&count,avg,n); // function for studnts > avg
 printf("Average marks=%.2f\n",avg);
 printf("No. of students with marks > avg is %d\n",count);
 return 0;
}
 void assign_mrk(int mark[N],int n)
{
 int i;
 for(i=0;i<n;i++)
 {
  mark[i]=rand()%101; // assign marks
 }
}
 void print_mrk(int mark[],int n)
{
 int i;
 printf("Marks of the students are:\n");
 for(i=0;i<n;i++)
 {
```

```
  printf("%4d",mark[i]);
  if((i+1)%10==0) // printing 10 marks in one line
  {
   printf("\n");
  }
 }
 printf("\n");

}
double cal_avg(int *mark,int n)
{
 int i,sum;
 sum=0;
 for(i=0;i<n;i++)  // fund sum of marks
 {
  sum +=mark[i];
 }
 return sum/(double) n;
}
void gt_avg(int *mark, int *count, double avg, int n)
{
 int i,cnt=0;
 for(i=0;i<n;i++)   // cal. no. of students > avg
 {
  if(mark[i]>avg)
  {
   cnt++;
  }
 }
 *count =cnt;
}
```

Listing 49: C Program "gtavg.c"

Here is the 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
```

Obviously, we get the same output. Noticeably, the `main()` mainly consists of functions call. Note that in each function call, we are passing `mark`, which is nothing but `&mark[0]`, the address of the first element. In the function prototype, we write this as `int [N]`, or `int []`, or `int *`. The first two is equivalent to `int *`, the last one. The first two have been provided in compiler for the convenience of a programmer. Similarly, `int mark[N]`, or `int mark[]`, or `int *mark` are used in functions definition. The first two again are equivalent to `int *mark`. Also, note that identifier `mark` used in the functions definition has nothing to do with students mark declared in `main()`. This `mark` in functions definition are local to functions and it is a pointer to type `int`.

As a final program, we sort the students' marks in decreasing order. For this, we use the bubble-sort algorithm. This algorithm works as follows.

1. Take the mark at the first position as the top element. Now compare the top element to the second element. If the second element is bigger, swap it with the top element.

2. Now compare the top element with the third element and swap with top element if the third element is bigger.

3. We continue this process until the top element is compared with the last element. The whole process is the first pass, and now the top element contains the largest element. This is like a bubble floating to the top.

4. Now start at the second top element and carry the second pass. As a result, the second top element contains the next largest element.

5. For `n` number of marks, they will be in a sorted order at the end of `n-1` pass.

The code is given below. We also use the swap function used in Listing 47. Also, as discussed earlier, we need to pass the addresses of the elements to be swapped.

```
#include<stdio.h>
#include<stdlib.h>
#define N 100
void assign_mrk(int [N],int);
void print_mrk(int [],int);
void bubble_srt(int *,int);
```

```
void swap(int *,int *);
int main()
{
 int mark[N],n;      // n actual student number
 printf("Enter actual no. of students: ");
 scanf("%d",&n);
 assign_mrk(mark,n);      //function to assign marks
 printf("Marks of the students are:\n");
 print_mrk(mark,n);       // function to print marks
 bubble_srt(mark,n);     // function to sort marks
 printf("Marks of the students in descendig order are:\n");
 print_mrk(mark,n);  // function to print marks
 return 0;
}
 void assign_mrk(int mark[N],int n)
{
 int i;
 for(i=0;i<n;i++)    // assign marks
 {
  mark[i]=rand()%101;
 }
}
 void print_mrk(int mark[],int n)
{
 int i;
 for(i=0;i<n;i++)
 {
  printf("%4d",mark[i]);
  if((i+1)%10==0)  // printing 10 marks in one line
  {
   printf("\n");
  }
 }
 printf("\n");
}
 void bubble_srt(int *mark,int n)
{
 int i,j;
 for(i=0;i<n-1;i++)       // n-1 passes
```

```
 {
  for(j=i+1;j<n;j++)  // compare mark[i] with mark[j],j=i+1,..,n-1
  {
   if(mark[j]>mark[i])
   {
    swap(&mark[i],&mark[j]);
   }
  }
 }
}
void swap(int *a,int *b)
{
 int t;
 t=*a;
 *a=*b;
 *b=t;
}
```

Listing 50: C Program "bubsrt.c"

Here is the 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
Marks of the students in descendig order are:
 100  99  98  88  88  85  83  83  81  80
  79  79  78  78  77  76  75  75  74  72
  71  69  67  65  63  62  61  61  61  59
  58  49  49  47  41  41  37  33  31  30
  29  27  25  23  23  22  19  18  17  16
  15  13  12  11   8   5   4
```

Return to Lecture Topics