Introduction to Scientific and Engineering Computation

(BIL 102E)

LECTURE 9

Pointers and Arrays

Pointers

So far we have been using variables to access the main memory. This is a pretty static way of reaching memory as we have to define variables when writing the program and cannot change these in run (execution) time.

Also with the types of variables it is not possible to handle large amounts of data.

Sometimes it is required to pass large amounts of data between functions, or allocate memory locations dynamically. In such situations, more flexible ways of handling the memory is necessary.

Pointers are used for this purpose.

Address-of Operator (&)

It is possible to find the actual address of a variable using the & operator. For example,

```
#include <stdio.h>
main() {
  int k=5;
  float x=3.14;
  printf("Value of k is %d\n",k);
  printf("Address of k is %p\n",&k);
  printf("Value of x is %f\n",x);
  printf("Address of x is %p\n",&x);
}
```

This finds the address of the variable k.

```
Value of k is 5
Address of k is 0065FDF4
Value of x is 3.140000
Address of x is 0065FDF0
```

Note that we have used %p to format the memory address.

The presentation of the address format can change from compiler to compiler.

Pointer Variables

A pointer is actually a memory address. (We can say that the pointer points to an address in memory).

We can define variables that are pointers. These variables allow us to reach the main memory directly.

A pointer variable can be defined by using an asterisks in front of the variable name. The syntax is

*data_type *pointer_name;*

For example,

int *pk;

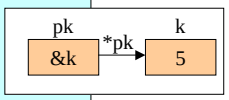
Defines a pointer variable pk which can point integer types of data in memory.

Dereferencing

The value of the memory location where a pointer points can be reached again by using an asterisks in front of the variable name. This is called dereferencing.

```
#include <stdio.h>
main() {
  int k=5;
  int *pk;
  pk = &k;
  printf("Value of k is %d\n",k);
  printf("Address of k is %p\n",&k);
  printf("Value of pk is %p\n",pk);
  printf("Address of pk is %p\n",&pk);
  printf("Value referenced by pk is %d\n",*pk);
}
```

Assign the address of k to pk.




```
Value of k is 5
Address of k is 0065FDF4
Value of pk is 0065FDF4
Address of pk is 0065FDF0
Value referenced by pk is 5
```

*pk means the value of the memory location pointed by pk.

It is possible to change the value of the memory location pointed by a pointer by the dereference operator (*).

```
#include <stdio.h>
main() {
  int j=5;
  int *pk = &j;
  *pk = 10;
  printf("Value of j is %d\n",j);
  printf("Address of j is %p\n",&j);
  printf("Value of pk is %p\n",pk);
  printf("Address of pk is %p\n",&pk);
  printf("Value referenced by pk is %d\n",*pk);
}
```

Declare and initialize pk.
The same as writing:
int *pk;
pk=&j;

Change the value of the memory location pointed by pk.

```
Value of j is 10
Address of j is 0065FDF4
Value of pk is 0065FDF4
Address of pk is 0065FDF0
Value referenced by pk is 10
```


Null Pointers

A pointer is said to be a null pointer when it points to address 0. A null pointer is assumed to be not pointing to a valid data address.

Null pointers can be used to test whether a pointer is assigned to a value.

For example,

```
int *p;
p = 0; /* Make p a null pointer */
if (p == 0) /* check whether p is a null pointer */
  printf("p is a null pointer");
```

We will see more examples on null pointers later in the course.

Arrays

Arrays are collection of similar data items. Arrays allow us to reach a set of memory locations of same type with the same name (by the help of an index).

Each item in an array is called an element.

The syntax for declaring an array is as follows:

data_type array_name[size];

This defines an array with name *array_name*. This array holds size number of elements of type *data_type*. Obviously *size* should be a constant integer expression.

For example,

int arr_x[3];

Declares an array named arr_x which holds 3 integer values.

Reaching elements of an array

It is possible to access each element of an array by using indices in square brackets.

The important point to remember is that all arrays in C are indexed starting at 0. That is, in the previous example arr_x[0] is the first element of the array and arr_x[2] is the last element of the array.

```
#include <stdio.h>
main() {
 float c[3];
 c[0] = 1.5;
 c[1] = c[0] + 2.5;
 c[2] = c[0] + c[1];
 printf("c[0]=%4.1f  c[1]=%4.1f c[2]=%4.1f\n",
        c[0],c[1],c[2]);
}
```

Declare an array of type float and of size 3.

Set the first element of the array to 1.5.

Note that we use c[0], c[1], c[2] as if they are variables of type **float**.

```
c[0]= 1.5  c[1]= 4.0 c[2]= 5.5
```


Example

Write a program that asks the number of students and then mark of each student in a class. The program should calculate and print out the arithmetic mean and the standard deviation of marks. Note that

$$\tilde{a} = \frac{\sum_{i=1}^{n} m_i}{n}$$

$$s = \sqrt{\frac{\sum_{i=1}^{n} (m_i - \tilde{a})^2}{n-1}}$$

A possible output of the program:

```
Number of students : 3
Mark of student 1 : 10
Mark of student 2 : 20
Mark of student 3 : 30
Average : 20.0
Standard Deviation : 10.0
```

where n indicates the number of students, m_i is the mark of student i, ã and s are arithmetic mean and standard deviation, respectively. Your program should assume that there could be at least 2 and at most 100 students in a class.


```
#include <stdio.h>
#include <math.h>
#define MAX_STUDS 100
main() {
  float marks[MAX_STUDS], sum=0, avg, std_dev;
  int i,num_of_studs;
  printf("Number of students : ");
  scanf("%d", &num_of_studs);
  if((num_of_studs<2) || (num_of_studs>MAX_STUDS)) {
    printf("Invalid number!\n");
       return 1;
  }
  for(i=0;i<num_of_studs;i++) {
     printf("Mark of student %d : ",i+1);
         scanf("%f",&marks[i]);
         sum += marks[i];
  }
  avg = sum/num_of_studs; /* Calculate the average mark */
  printf("Average : %4.1f\n",avg);
  sum=0;
  for(i=0;i<num_of_studs;i++)
     sum += pow(marks[i]-avg,2);
  std_dev=sqrt(sum/(num_of_studs-1));
  printf("Standard Deviation : %4.1f\n",std_dev);
  return 0;
}
```

This is a compiler directive that defines a *macro*. The compiler will replace MAX_STUDS with 100 in the program.

Check whether the number of students are in the acceptable range.

Read mark of each student into an array and accumulate the sum of marks in the variable sum.

Accumulate the sums of the squares of difference of marks from the average in the variable sum.

Initializing Arrays

It is possible to initialize arrays by using a list of values enclosed in braces ({ and }). For example,

```
int ak[3] = { 1, 5, 9 };
```

Declares an array named ak with 3 elements and the elements are initialized to 1, 5 and 9. This is same as writing the following code:

```
int ak[3];
ak[0] = 1;
ak[1] = 5;
ak[2] = 9;
```

If an array is initialized we do not have to give the size of the array. For example, we could have written:

```
int ak[] = { 1, 5, 9 };
```

Size of an Array

The number of bytes reserved in memory by an array can be found by using the **sizeof()** function.

For example,

```
int ak[] = { 1, 5, 9 };
printf("Memory required by ak is %d bytes\n",
  sizeof(ak));
printf("Hence ak has %d elements\n",
  sizeof(ak)/sizeof(int));
```

```
Memory required by ak is 12 bytes
Hence ak has 3 elements
```


Arrays and Pointers

Array variables can be considered as pointers that point to the beginning of the array. Therefore, it is possible to assign an array variable to a pointer directly. For example,

```
#include <stdio.h>
main() {
  int arr_k[3]={2,3,7};
  int *ptr_k;
  ptr_k=arr_k;
  printf("The first element of the array arr_k is %d\n",*ptr_k);
}
```

Note that we do not use the address of (&) operator to get the address of the array.

The pointer points to the first element of the array.

```
The first element of the array arr_k is 2
```

Some compilers allow pointers to be used like arrays when referencing the elements:

```
#include <stdio.h>
main() {
  int i, arr_k[3]={2,3,7};
  int *ptr_k=arr_k;
  for(i=0;i<3;i++)
    printf("arr_k[%d] = %d\n",i, ptr_k[i]);
}
```

```
arr_k[0] = 2
arr_k[1] = 3
arr_k[2] = 7
```


Pointer Arithmetic

By adding or subtracting integers from pointers it is possible to have the pointers point to other elements in an array. For example, if *ptr_k points to the first element of an array *(ptr_k+1) points to the second element. The following programs produce exactly the same result as the previous example.

```
#include <stdio.h>
main() {
  int i, arr_k[3]={2,3,7};
  int *ptr_k=arr_k;
  for(i=0;i<3;i++)
    printf("arr_k[%d] = %d\n",i, *(ptr_k+i));
}
```

Note that ptr_k+1 does not necessarily point to next byte in the memory. It points to next element.

This is different than *ptr_k+i, which would produce 2, 3, 4 for this example.

```
arr_k[0] = 2
arr_k[1] = 3
arr_k[2] = 7
```

```
#include <stdio.h>
main() {
  int i, arr_k[3]={2,3,7};
  int *ptr_k=arr_k;
  for(i=0;i<3;i++,ptr_k++)
    printf("arr_k[%d] = %d\n",i,*ptr_k);
}
```

ptr_k++ makes ptr_k to point the next element in the array.

```
arr_k[0] = 2
arr_k[1] = 3
arr_k[2] = 7
```


Considerations

When using pointers and arrays extreme precautions must be taken.

It is possible to reach memory locations that are not reserved for our program and change them by the help of pointers.

This may result in unexpected outputs and even **halt the computer**.

Possible Errors:

Use of uninitialized pointers:

```
int *ptr_k;
int k=5;
*ptr_k = k;
/* Problem: I have used the pointer before having it
point to a meaningful address */
```

Indices that are out of range:

```
int j=5,arr_x[3];
int *ptr_k=arr_x;
arr_x[-1]=1; /* Index is out of range */
arr_x[j]=7; /* Index is out of range */
*(ptr_k+j) = 8;/*Pointer points to undesired location */
```


Pointers as arguments to functions

Like other types of data pointers can be used as arguments of functions. This could be useful if the function is required to change some variables in the calling function or large amounts of data is to be transferred to the function.

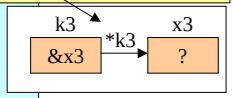
We have been already passing the addresses of variables to the scanf function, which reads data from the keyboard and changes the variables accordingly. The following is a simple example:

```
#include <stdio.h>
int Times23(int k, int *k3) {
  *k3 = k * 3;
  return k * 2;
}
main() {
  int x=5,x2,x3;
  x2=Times23(x, &x3);
  printf("x = %d\nx2 = %d\nx3 = %d\n",x,x2,x3);
}
```

The argument k3 is a pointer to integers.

We have to send an address here.

When the function is called:




```
x = 5
x2 = 10
x3 = 15
```


It is possible to use pointer arguments to pass large amounts of data to and from functions (by means of arrays). In the following, a function that calculates the sum of the first n elements of an array is given.

```
#include <stdio.h>
int sum(int *arr, int n) {
  int i,sm=0;
  for (i=0; i<n; i++)
        sm+=arr[i];
  return sm;
}
main() {
       int i,x[]={1,3,3,7,8};
       for(i=1;i<6;i++)
         printf("sum(x,%d) = %d\n",i,sum(x,i));
}
```

Note that this function assumes *n* is less than or equal to the size of the array.

```
sum(x,1) = 1
sum(x,2) = 4
sum(x,3) = 7
sum(x,4) = 14
sum(x,5) = 22
```


Write a function called fswap that swaps the values of two floating point arguments:

```
void fswap(float *num1, float *num2){
  float temp;
  temp=*num1;
  *num1=*num2;
  *num2=temp;
}
```

Write a function called fsort2_asc that sorts the values of two floating point arguments in an ascending order:

```
void fsort2_asc(float *num1, float *num2){
  if(*num1 > *num2)
        fswap(num1,num2);
}
```

Example

Write a program that asks the marks of students in a class until an invalid mark is entered and then sorts the marks in an ascending order.

Analysis:

1. In an infinite loop the marks can be read and put into an integer array *mark*. The number of students can be hold in a variable named *num_studs*.
2. In order to sort the array a **for** loop where an integer *i* is used to count from 1 to *num_studs-1* can be used. In each loop *ith* smallest element of the array is found and put into its final place (i.e. the *ith*. element).
3. So as to find the *ith* smallest element a second (inner) **for** loop can be used. In this do loop, the counter (j) starts from *i+1* and always ensures that the smallest element encountered that far is replaced in *ith* element.


```c
#include <stdio.h>
#define MAX_STUDS 100
main()
{
  int i,j, num_studs=0;
  float mark[MAX_STUDS];

  for(;;) {
     printf("What is the mark of Student %d?\n",num_studs+1);
     scanf("%f",&mark[num_studs]);
     if(mark[num_studs]<0.0 || mark[num_studs]>100.0)
        break;
        num_studs++;
  }
  for(i=0;i<num_studs-1;i++)
         for(j=i+1; j<num_studs; j++)
                fsort2_asc(&mark[i], &mark[j]);

  /* Print out the result */
  for(i=0;i<num_studs;i++)
         printf("%4.1f\n",mark[i]);
}
```


A Possible Output of the Program

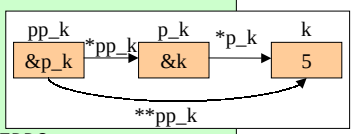
```
What is the mark of Student 1?
56
What is the mark of Student 2?
89
What is the mark of Student 3?
76
What is the mark of Student 4?
34
What is the mark of Student 5?
45
What is the mark of Student 6?
67
What is the mark of Student 7?
-1
34.0
45.0
56.0
67.0
76.0
89.0
```


Pointers to Pointers

It is possible to define a pointer that points to another pointer. For example,

```c
int k=5;
int *p_k = &k;
int **pp_k = &p_k; /*This defines a pointer to another pointer*/
printf("Value of k is %d\n",k);
printf("Address of k is %d\n",&k);
printf("Value of p_k is %p\n",p_k);
printf("Address of p_k is %p\n",&p_k);
printf("Value referenced by p_k is %d\n",*p_k);
printf("Value of pp_k is %p\n",pp_k);
printf("Value referenced by pp_k is %p\n",*pp_k);
printf("Value referenced by value referenced by pp_k is %d\n",
 **pp_k);
```

```
Value of k is 5
Address of k is 6684124
Value of p_k is 0065FDDC
Address of p_k is 0065FDD8
Value referenced by p_k is 5
Value of pp_k is 0065FDD8
Value referenced by pp_k is 0065FDDC
Value referenced by value referenced by pp_k is 5
```



Arrays of Pointers

It is possible to define an array of pointers using the following syntax:

data_type *arr_ptr[n];

For example,

```
#include <stdio.h>
main() {
  int i;
  float x1 = 1.5;
  float x2 = 2.7;
  float x3 = 3.9;
  float *ptr_ax[3]={&x1,&x2,&x3};
  for(i=0;i<3;i++)
        printf("*ptr_ax[%d] = %4.1f\n",i,*ptr_ax[i]);
}
```

```
*ptr_ax[0] =  1.5
*ptr_ax[1] =  2.7
*ptr_ax[2] =  3.9
```


Pointing to Functions

Remember that both data and instructions are hold in the main memory. As it is possible to have pointers that point to data elements it is also possible to have pointers that point to instructions (functions).

In order to define a pointer to a function, we have to define what kind of function the pointer points to.

Functions are categorized according to their interface (That is, the type of output they produce and the number and the types of arguments they accept).

For example,

```
double (*ptrf)(int x, char ch);
```

Declares a pointer named ptrf that can point to functions that produce results in type **double** and have two arguments the first of which is an integer and the second of which is **char.**


```
long int Factorial(int k) {
  long int fact=1,i;
  for (i=2; i<=k; i++)
    fact *= i;
  return fact;
}
long int Fibonacci(int n) {
        long int Fi=1, F1=1, F2=0,i;
        if(n<0)
                Fi=-1;
        else if(n==0)
                Fi=0;
        else
                for(i=2; i<=n; i++){
                        Fi=F1+F2;
                        F2=F1;
                        F1=Fi;
                }
        return Fi;
}
void Printfx(long int (*f)(int t), int x) {
  printf("f(%d)=%ld\n",x,f(x));
}
```

Fibonacci numbers are defined as follows:

$f_0=0$

$f_1=1$

$f_k= f_{k-1}+ f_{k-2}$ (for k>1)

```
Calling Factorial :
f(4)=24
Calling Fibonacci :
f(4)=3
```

```
main() {
  printf("Calling Factorial :\n");
  Printfx(Factorial,4);
  printf("Calling Fibonacci :\n");
  Printfx(Fibonacci,4);
}
```


Example

Derivative of a real valued function f(x) at a given point x on the real axis can be defined as

$$\frac{\partial f(x)}{\partial x} = \lim_{\varepsilon \to 0^+} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Hence, an approximation is given by

$$\frac{\partial f(x)}{\partial x} = \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

where ϵ is a small number.

Write a function called D(f, x0) that finds the derivative of a function f(x) at x.

```c
#include <stdio.h>
#include <math.h>
#define EPS 1.0e-5
#define PI 3.1415926
double D(double (*f)(double x), double point) {
  double diff;
  diff = ((*f)(point) - (*f)(point-EPS)) / EPS;
  return diff;
}
main() {
    double x,radx;
        for(x=0.0;x<=90;x+=15.0){
          radx = x * PI/180.0;
          printf("D[sin(%2.0f)] = %5.3f   cos(%2.0f) = %5.3f\n",
                  x,D(sin,radx),x,cos(radx));
        }
}
```

This function takes derivative of **any function** which has a double argument and produces a double result.

```
D[sin( 0)] = 1.000   cos( 0) = 1.000
D[sin(15)] = 0.966   cos(15) = 0.966
D[sin(30)] = 0.866   cos(30) = 0.866
D[sin(45)] = 0.707   cos(45) = 0.707
D[sin(60)] = 0.500   cos(60) = 0.500
D[sin(75)] = 0.259   cos(75) = 0.259
D[sin(90)] = 0.000   cos(90) = 0.000
```

We confirm that derivative of sinus is cosine.

Multidimensional Arrays

It is possible to define multi-dimensional arrays (arrays of arrays) by using the following syntax:

data_type *array_name*[size1][size2]...[sizeN];

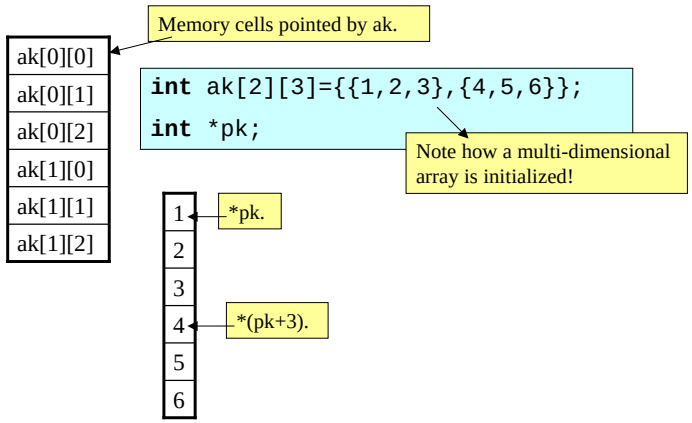
For example,

int ak[2][3];

Declares an array of type int, which has 2 elements, each of which can be seen as another array of type int with 3 elements. We might consider this as a 2 by 3 table:

ak[0][0]	ak[0][1]	ak[0][2]
ak[1][0]	ak[1][1]	ak[1][2]

Note that multi-dimensional arrays are stored in the memory in a rows first manner. In the above example,




```c
#include <stdio.h>
main(){
  int ak[][3]={{1,2,3},{4,5,6}};
  int i,j;
  for(i=0;i<2;i++)
  {
    for(j=0;j<3;j++)
          printf("%d \t",ak[i][j]);
        printf("\n");
  }
}
```

Note that only one dimension size can be omitted when initializing the array.

\t is the tab character.

```
1       2       3
4       5       6
```

```c
#include <stdio.h>
main(){
  int ak[][3]={{1,2,3},{4,5,6}};
  int i,j,*pk;
  pk=ak;
  for(i=0;i<2;i++)
  {
    for(j=0;j<3;j++,pk++)
          printf("%d \t",*pk);
        printf("\n");
  }
}
```

The same example where a pointer is used to reach the elements of the array.

```
1       2       3
4       5       6
```

Multi-dimensional arrays as arguments to functions

It is possible to use multi-dimensional arrays as arguments to functions. For example,

```
#include <stdio.h>
void PrintArray(int arr[][2], int n){
  int i,j;
  for(i=0; i<n; i++) {
    for(j=0; j<2; j++)
      printf("%d\t",arr[i][j]);
      printf("\n");
  }
}
main(){
       int Mat1[3][2]={{1,2},{3,4},{5,6}};
       PrintArray(Mat1,3);
}
```

This prints out any 2 column array with n rows in a nice tabular format.

Note that we have to give the number of rows for the indexing to work.

```
1       2
3       4
5       6
```


We can use pointers for the same purpose.

Note that this is more flexible, as this will print out any *n* by *m* matrix.

```
#include <stdio.h>
void PrintMatrix(int *mat, int n, int m) {
  int i,j;
  for(i=0; i<n; i++) {
    for(j=0; j<m; j++)
      printf("%d\t",*(mat+i*m+j));
      printf("\n");
  }
}
main(){
       int Mat1[3][2]={{1,2},{3,4},{5,6}};
       PrintMatrix(Mat1,3,2);
}
```

```
1       2
3       4
5       6
```

Pointer arithmetic is used to find the i,j th element of the matrix.

The following function adds 2 n by m matrices:

```
void MatrixAdd(int *result, int *M1, int *M2, int n, int m) {
  int i,j;
  for(i=0; i<n; i++)
    for(j=0; j<m; j++)
         *(result+i*m+j) = *(M1+i*m+j) + *(M2+i*m+j);
}
```

For example,

```
main(){
       int Mat1[3][2]={{1,2},{3,4},{5,6}};
       int Mat2[3][2]={{3,3},{4,4},{5,5}};
       int Mat3[3][2];
       MatrixAdd(Mat3,Mat1,Mat2,3,2);
    PrintMatrix((int *) Mat3,3,2);
}
```

```
4       5
7       8
10      11
```


This does the same thing: This time we get the size of the matrix instead of dimensions;

```
void MatrixAdd2(int *result, int *Mat1, int *Mat2, int nm) {
  int i;
  int *r=result, *M1=Mat1, *M2=Mat2;
  for(i=0; i<nm; i++, r++, M1++, M2++)
    *r = *M1 + *M2;
}
```

For example,

```
main(){
    int Mat1[3][2]={{1,2},{3,4},{5,6}};
    int Mat2[3][2]={{3,3},{4,4},{5,5}};
    int Mat3[3][2];
    MatrixAdd(Mat3,Mat1,Mat2,3*2);
    PrintMatrix((int *) Mat3,3,2);
}
```

```
4       5
7       8
10      11
```