Pointers and Arrays

Pointers and Arrays

- When an array is declared,
  - The compiler allocates sufficient amount of storage to contain all the elements of the array in contiguous memory locations
  - The base address is the location of the first element (index 0) of the array
  - The compiler also defines the array name as a constant pointer to the first element

Example

- Consider the declaration:

  int x[5] = {1, 2, 3, 4, 5};

- Suppose that each integer requires 4 bytes
- Compiler allocates a contiguous storage of size 5x4 = 20 bytes
- Suppose the starting address of that storage is 2500

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Contd.

- The array name x is the starting address of the array
  - Both x and &x[0] have the value 2500
  - x is a constant pointer, so cannot be changed
    - X = 3400, x++, x += 2 are all illegal
- If int *p is declared, then

  p = x; and p = &x[0]; are equivalent
- We can access successive values of x by using p++ or p-- to move from one element to another

- Relationship between p and x:

```
p     =  &x[0]  =  2500
p+1   =  &x[1]  =  2504
p+2   =  &x[2]  =  2508
p+3   =  &x[3]  =  2512
p+4   =  &x[4]  =  2516
```

In general, *(p+i) gives the value of x[i]

- C knows the type of each element in array x, so knows how many bytes to move the pointer to get to the next element

Example: function to find average

```
int main()
{
  int x[100], k, n;

  scanf (“%d”, &n);

  for (k=0; k<n; k++)
     scanf (“%d”, &x[k]);

  printf  (“\nAverage is %f”,
                 avg (x, n));
  return 0;
}
```

```
float avg (int array[], int size)
{
  int  *p, i , sum = 0;

  p = array;

  for (i=0; i<size; i++)
      sum = sum + *(p+i);

  return ((float) sum / size);
}
```

The pointer p can be subscripted also just like an array!

```
int main()
{
  int x[100], k, n;

  scanf (“%d”, &n);

  for (k=0; k<n; k++)
     scanf (“%d”, &x[k]);

  printf  (“\nAverage is %f”,
                avg (x, n));
  return 0;
}
```

```
float avg (int array[], int size)
{
  int  *p, i , sum = 0;

  p = array;

  for (i=0; i<size; i++)
      sum = sum + p[i];

  return ((float) sum / size);
}
```

Important to remember

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Consequence: While traversing the elements of an array (either using [ ] or pointer arithmetic), we can accidentally access off the end of an array (access more elements than what is there in the array)
  - Consequence: We must pass the array and its size to a function which is going to traverse it, or there should be some way of knowing the end based on the values (Ex., a –ve value ending a string of +ve values)
- Accessing arrays out of bound can cause segmentation faults
  - Hard to debug (already seen in lab)
  - Always be careful when traversing arrays in programs

Pointers to Structures

Pointers to Structures

- Pointer variables can be defined to store the address of structure variables
- Example:

```
struct student {
        int   roll;
        char  dept_code[25];
        float cgpa;
      };
struct student *p;
```

- Just like other pointers, p does not point to anything by itself after declaration
  - Need to assign the address of a structure to p
  - Can use & operator on a struct student type variable
  - Example:

```
struct student x, *p;
scanf("%d%s%f", &x.roll, x.dept_code, &x.cgpa);
p = &x;
```

- Once p points to a structure variable, the members can be accessed in one of two ways:
  - (*p).roll, (*p).dept_code, (*p).cgpa
    - Note the ( ) around *p
  - p –> roll, p –> dept_code, p –> cgpa
    - The symbol –> is called the arrow operator
- Example:
  - printf("Roll = %d, Dept.= %s, CGPA = %f\n", (*p).roll, (*p).dept_code, (*p).cgpa);
  - printf("Roll = %d, Dept.= %s, CGPA = %f\n", p->roll, p->dept_code, p->cgpa);

Pointers and Array of Structures

- Recall that the name of an array is the address of its 0-th element
  - Also true for the names of arrays of structure variables.
- Consider the declaration:

```
struct student class[100],  *ptr ;
```

- The name class represents the address of the 0-th element of the structure array
  - ptr is a pointer to data objects of the type struct student
- The assignment

  ptr = class;

  will assign the address of class[0] to ptr
- Now ptr->roll is the same as class[0].roll. Same for other members
- When the pointer ptr is incremented by one (ptr++) :
  - The value of ptr is actually increased by sizeof(struct student)
  - It is made to point to the next record
  - Note that sizeof operator can be applied on any data type

A Warning

- When using structure pointers, be careful of operator precedence
  - Member operator "." has higher precedence than "*"
    - ptr –> roll and (*ptr).roll mean the same thing
    - *ptr.roll will lead to error
  - The operator "–>" enjoys the highest priority among operators
    - ++ptr –> roll will increment ptr->roll, not ptr
    - (++ptr) –> roll will access (ptr + 1)->roll (for example, if you want to print the roll no. of all elements of the class array)