Pointers and Arrays

Lecture 27

- 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:
  
  ```c
  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

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>1</td>
<td>2500</td>
</tr>
<tr>
<td>x[1]</td>
<td>2</td>
<td>2504</td>
</tr>
<tr>
<td>x[2]</td>
<td>3</td>
<td>2508</td>
</tr>
<tr>
<td>x[3]</td>
<td>4</td>
<td>2512</td>
</tr>
<tr>
<td>x[4]</td>
<td>5</td>
<td>2516</td>
</tr>
</tbody>
</table>

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
  
  ```c
  p = x;  \quad \text{and} \quad 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 \( x \):

\[
\begin{align*}
p &= &x[0] &= 2500 \\
p+1 &= &x[1] &= 2504 \\
p+2 &= &x[2] &= 2508 \\
p+3 &= &x[3] &= 2512 \\
\end{align*}
\]

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

```c
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!

```c
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 strange problems
  - Very hard to debug
  - Always be careful when traversing arrays in programs
2D Array

In Memory:

2D Array

In Memory:

|---------|---------|---------|---------|---------|---------|---------|---------|---------|

A
2D Array

A

*(A+0)

A[0]

A[1]

A[2]

In Memory:


*(A+1)

*(A+2)

2D Array

A

*(A+0)

A[0]

A[1]

A[2]

In Memory:

2D Array

In general, $A[i][j] = *(A+i)+j$
2D Array

In general, \( A[i][j] = *(A+i)+j \)

In Memory:

\[
\begin{array}{cccc}
A[0][0] & A[0][1] & A[0][2] \\
*(A+0)+0 & *(A+0)+1 & *(A+0)+2 \\
*(A+1)+0 & *(A+1)+1 & *(A+1)+2 \\
*(A+2)+0 & *(A+2)+1 & *(A+2)+2 \\
\end{array}
\]
Pointers to Structures

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

  ```c
  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:

    ```c
    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:

- \((^p).\text{roll}, (^p).\text{dept\_code}, (^p).\text{cgpa}\)
  - Note the ( ) around \(^p\)
- \( p \rightarrow \text{roll}, p \rightarrow \text{dept\_code}, p \rightarrow \text{cgpa}\)
  - The symbol \( \rightarrow \) is called the arrow operator

Example:

- \( \text{printf("Roll = %d, Dept.= %s, CGPA = %f\n", (^p).roll, (^p).dept\_code, (^p).cgpa);} \)
- \( \text{printf("Roll = %d, Dept.= %s, CGPA = %f\n", p->\text{roll}, p->\text{dept\_code}, p->\text{cgpa});} \)

---

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:

\[
\text{struct student class}[100], *\text{ptr} ;
\]
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
  
  ```c
  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

```c
struct student {
    char name[20];
    int roll;
}

int main()
{
    struct student class[50], *p;
    int i, n;
    scanf("%d", &n);
    for (i=0; i<n; i++)
        scanf("%s%d", class[i].name, &class[i].roll);
    p = class;
    for (i=0; i<n; i++) {
        printf("%s  %d
", class[i].name, class[i].roll);
        printf("%s  %d
", *(p+i).name, *(p+i).roll);
        printf("%s  %d
", (p+i)->name, (p+i)->roll);
        printf("%s  %d
", p[i].name, p[i].roll);
    }
}
```

```
3
Ajit 1001
Abhishek 1005
Riya 1007
Ajit 1001
Ajit 1001
Ajit 1001
Ajit 1001
Abhishek 1005
Abhishek 1005
Abhishek 1005
Abhishek 1005
Riya 1007
Riya 1007
Riya 1007
Riya 1007
```
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)
- When not sure, use ( and ) to force what you want

Practice Problems

- Look at all problems you have done earlier on arrays (including arrays of structures). Now rewrite all of them using equivalent pointer notations
  - Example: If you had declared an array
    - `int A[50]`
    - Now do
    - `int A[50], *p;`
    - `p = A;`
    - and then write the rest of the program using the pointer `p` (without using [ ] notation)
Problem with Arrays

- Sometimes
  - Amount of data cannot be predicted beforehand
  - Number of data items keeps changing during program execution
- Example: Search for an element in an array of N elements
- One solution: find the maximum possible value of N and allocate an array of N elements
  - Wasteful of memory space, as N may be much smaller in some executions
  - Example: maximum value of N may be 10,000, but a particular run may need to search only among 100 elements
    - Using array of size 10,000 always wastes memory in most cases
Better Solution

- Dynamic memory allocation
  - Know how much memory is needed after the program is run
    - Example: ask the user to enter from keyboard
  - Dynamically allocate only the amount of memory needed
- C provides functions to dynamically allocate memory
  - `malloc`, `calloc`, `realloc`

Memory Allocation Functions

- `malloc`
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space
- `calloc`
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- `free`
  - Frees previously allocated space.
- `realloc`
  - Modifies the size of previously allocated space.
- We will only do `malloc` and `free`
Allocating a Block of Memory

- A block of memory can be allocated using the function `malloc`
  - Reserves a block of memory of specified size and returns a pointer of type `void`
  - The return pointer can be type-casted to any pointer type
- General format:
  
  
  ```c
  type *p;
p = (type *) malloc (byte_size);
  ```

Example

```c
p = (int *) malloc(100 * sizeof(int));
```

- A memory space equivalent to 100 times the size of an int bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer `p` of type `int`
Contd.

- `cptr = (char *) malloc (20);`
  
  Allocates 20 bytes of space for the pointer `cptr` of type `char`

- `sptr = (struct stud *) malloc(10*sizeof(struct stud));`
  
  Allocates space for a structure array of 10 elements. `sptr` points to a structure element of type `struct stud`

  *Always use `sizeof` operator to find number of bytes for a data type, as it can vary from machine to machine.*

---

Points to Note

- `malloc` always allocates a block of contiguous bytes
  - The allocation can fail if sufficient contiguous memory space is not available
  - If it fails, `malloc` returns `NULL`

```c
if (((p = (int *) malloc(100 * sizeof(int))) == NULL))
{
    printf("Memory cannot be allocated");
    exit();
}
```
Using the malloc’ed Array

- Once the memory is allocated, it can be used with pointers, or with array notation
- Example:

```c
int *p, n, i;
scanf("%d", &n);
p = (int *) malloc (n * sizeof(int));
for (i=0; i<n; ++i)
    scanf("%d", &p[i]);
```

The n integers allocated can be accessed as *p, *(p+1), *(p+2),…, *(p+n-1) or just as p[0], p[1], p[2], …, p[n-1]

Example

```c
int main()
{
    int i,N;
    float *height;
    float sum=0,avg;

    printf("Input no. of students\n");
    scanf("%d", &N);

    height = (float *)
        malloc(N * sizeof(float));

    for(i=0;i<N;i++)
        scanf("%f", &height[i]);

    for(i=0;i<N;i++)
        sum += height[i];

    avg = sum / (float) N;

    printf("Average height = %f \n", avg);
    free (height);
    return 0;
}
```

```c
printf("Input heights for \n");
for (i=0; i<N; i++)
    scanf("%f", &height[i]);

for(i=0;i<N;i++)
    sum += height[i];

avg = sum / (float) N;

printf("Average height = %f\n", avg);
```
Releasing the Allocated Space: free

- An allocated block can be returned to the system for future use by using the free function
- General syntax:
  ```c
  free (ptr);
  ```
  where `ptr` is a pointer to a memory block which has been previously created using `malloc`
- Note that no size needs to be mentioned for the allocated block, the system remembers it for each pointer returned

Can we allocate only arrays?

- `malloc` can be used to allocate memory for single variables also
  ```c
  p = (int *) malloc (sizeof(int));
  ```
  Allocates space for a single int, which can be accessed as `*p`
- Single variable allocations are just special case of array allocations
  ```c
  Array with only one element
  ```
malloc( )-ing array of structures

```c
typedef struct{
    char name[20];
    int roll;
    float SGPA[8], CGPA;
} person;

int main() {
    person *student;
    int i,j,n;
    scanf("%d", &n);
    student = (person *)malloc(n*sizeof(person));
    for (i=0; i<n; i++) {
        scanf("%s", student[i].name);
        scanf("%d", &student[i].roll);
        for(j=0; j<8; j++) scanf("%f", &student[i].SGPA[j]);
        scanf("%f", &student[i].CGPA);
    }
    return 0;
}
```

Static array of pointers

```c
#define N 20
#define M 10

int main() {
    char word[N], *w[M];
    int i, n;
    scanf("%d", &n);
    for (i=0; i<n; ++i) {
        scanf("%s", word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy (w[i], word);
    }
    for (i=0; i<n; i++) printf("w[%d] = %s
", i, w[i]);
    return 0;
}
```
Static array of pointers

```c
#define N 20
#define M 10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        scanf("%s", &word);
        w[i] = (char *) malloc ((strlen(word) + 1) * sizeof(char));
        strcpy (w[i], word);
    }
    for (i = 0; i < n; i++) printf("w[%d] = %s \n", i, w[i]);
    return 0;
}
```

Output:
```
4
Tendulkar
Sourav
Khan
India
w[0] = Tendulkar
w[1] = Sourav
w[2] = Khan
w[3] = India
```

How it will look like

```
  w
 /|
0 | Tendulkar
 /|
1 | Sourav
 /|
2 | Khan
 /|
3 | India
 /|
9 |
```
Pointers to Pointers

- Pointers are also variables (storing addresses), so they have a memory location, so they also have an address
- Pointer to pointer – stores the address of a pointer variable

```c
int x = 10, *p, **q;
p = &x;
q = &p;
printf("%d %d %d", x, *p, *(*q));
```

will print 10 10 10 (since *q = p)

Allocating Pointer to Pointer

```c
int **p;
p = (int **) malloc(3 * sizeof(int *));
```

```
  p  |  p[0]  |
    |   int * |
    |   int * |
  p[1] |   int * |
  p[2] |   int * |
```
Dynamic Arrays of pointers

```c
int main()
{
    char word[20], **w;  /* **w is a pointer to a pointer array */
    int i, n;
    scanf("%d", &n);
    w = (char **) malloc (n * sizeof(char *));
    for (i=0; i<n; ++i) {
        scanf("%s", word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy(w[i], word);
    }
    for (i=0; i<n; i++) printf("w[%d] = %s \n", i, w[i]);
    return 0;
}
```

Output

```
India
Australia
Kenya
NewZealand
SriLanka
```

w[0] = India
w[1] = Australia
w[3] = NewZealand
w[4] = SriLanka
Dynamic Allocation of 2-d Arrays

```c
int **allocate (int h, int w)
{
    int **p;
    int i, j;
    p = (int **) malloc(h*sizeof (int *) );
    for (i=0;i<h;i++)
        p[i] = (int *) malloc(w * sizeof (int));
    return(p);
}

void read_data (int **p, int h, int w)
{
    int i, j;
    for (i=0;i<h;i++)
        for (j=0;j<w;j++)
            scanf ("%d", &p[i][j]);
}
```

Elements accessed like 2-D array elements.
void print_data (int **p, int h, int w)
{
  int i, j;
  for (i=0; i<h; i++)
  {
    for (j=0; j<w; j++)
      printf ("%5d ", p[i][j]);
    printf ("\n");
  }
}

Contd.

int main()
{
  int **p;
  int M, N;
  printf ("Give M and N \n");
  scanf ("%d%d", &M, &N);
  p = allocate (M, N);
  read_data (p, M, N);
  printf ("\nThe array read as \n");
  print_data (p, M, N);
  return 0;
}

Give M and N
3 3
1 2 3
4 5 6
7 8 9
The array read as
1 2 3
4 5 6
7 8 9
Memory Layout in Dynamic Allocation

```c
int **allocate (int h, int w)
{
    int **p;
    int i, j;
    p = (int **)malloc(h*sizeof (int *));
    for (i=0; i<h; i++)
    {
        p[i] = (int *)malloc(w*sizeof(int));
    }
    return(p);
}
```

```c
int main()
{
    int **p;
    int M, N, i, j;
    printf ("Give M and N \n");
    scanf ("%d%d", &M, &N);
    p = allocate (M, N);
    for (i=0;i<M;i++)
    {
        for (j=0;j<N;j++)
        {
            printf("%u", &p[i][j]);
            printf("\n");
        }
    }
    return 0;
}
```

Output

```
3 3
31535120 31535128 31535136
31535152 31535156 31535160
31535184 31535188 31535192
31535216 31535220 31535224
```

Starting address of each row, contiguous (pointers are 8 bytes long)

Elements in each row are contiguous
Practice Problems

- Take any of the problems you have done so far using 1-d arrays or 2-d arrays. Now do them by allocating the arrays dynamically first instead of declaring them statically.