

Lecture 14

Dynamic Memory Allocation

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`

Dynamic memory allocation in C

- **Dynamic memory allocation** is the allocation of memory storage for use in a computer program during the runtime of that program.
- Memory is typically allocated from a large pool of all available unused memory called the *heap*, but may also be allocated from multiple pools.

Dynamic Memory Allocation

- Dynamic memory allocation is done during **runtime**.
- We determine the size of memory area to be allocated and the time, when allocation is done.
- Then we can allocate space as much as we need and just when we need it.
- When we no longer need it, we free it.

Dynamic Memory Allocation

- A dynamically allocated object remains allocated until it is deallocated explicitly, either by the programmer or by an garbage collector;

Garbage Collector

- In computing, **garbage collection** is a system of automatic memory management which seeks to reclaim memory used by objects which will never be referenced in the future.
- It is commonly abbreviated as GC.
- The part of a system which performs garbage collection is called a **garbage collector**.

Garbage Collector

- When a system has a garbage collector it is usually part of the language run-time system and integrated into the language.
- The language is said to be garbage collected.
- Garbage collection was invented by John McCarthy as part of the first Lisp system.

The basic principle of how a garbage collector works is:

- 1. Determine what data objects in a program cannot be referenced in the future
- 2. Reclaim the storage used by those objects

STATIC VS DYNAMIC MEMORY

Static memory allocation	Dynamic memory allocation
<p>In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.</p>	<p>In dynamic memory allocation, memory is allocated while executing the program. That means at run time.</p>
<p>Memory size can't be modified while execution. Example: array</p>	<p>Memory size can be modified while execution. Example: Linked list</p>

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.

SYNTAX

Function	Syntax
malloc ()	malloc (number *sizeof(int));
calloc ()	calloc (number, sizeof(int));
realloc ()	realloc (pointer_name, number * sizeof(int));
free ()	free (pointer_name);

MALLOC() FUNCTION IN C:

- `malloc ()` function is used to allocate space in memory during the execution of the program.
- `malloc ()` does not initialize the memory allocated during execution. It carries garbage value.
- `malloc ()` function returns null pointer if it couldn't able to allocate requested amount of memory.

MALLOC() FUNCTION IN C:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7      char *mem_allocation;
8      /* memory is allocated dynamically */
9      mem_allocation = malloc( 20 * sizeof(char) );
10     if( mem_allocation== NULL )
11     {
12         printf("Couldn't able to allocate requested memory\n");
13     }
14     else
15     {
16         strcpy( mem_allocation,"fresh2refresh.com");
17     }
18     printf("Dynamically allocated memory content : " \
19           "%s\n", mem_allocation );
20     free(mem_allocation);
21 }
```

CALLOC() FUNCTION IN C:

- `calloc ()` function is also like `malloc ()` function.
- But `calloc ()` initializes the allocated memory to zero. But, `malloc()` doesn't.

CALLOC() FUNCTION IN C:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7      char *mem_allocation;
8      /* memory is allocated dynamically */
9      mem_allocation = calloc( 20, sizeof(char) );
10     if( mem_allocation== NULL )
11     {
12         printf("Couldn't able to allocate requested memory\n");
13     }
14     else
15     {
16         strcpy( mem_allocation, "fresh2refresh.com");
17     }
18     printf("Dynamically allocated memory content : " \
19           "%s\n", mem_allocation );
20     free(mem_allocation);
21 }
```


REALLOC() FUNCTION IN C:

- `realloc ()` function modifies the allocated memory size by `malloc ()` and `calloc ()` functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

FREE() FUNCTION IN C:

- `free ()` function frees the allocated memory by `malloc ()`, `calloc ()`, `realloc ()` functions and returns the memory to the system.

Malloc Vs Calloc

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
<pre>int *ptr; ptr = malloc(20 * sizeof(int));</pre> For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	<pre>int *ptr; Ptr = calloc(20, 20 * sizeof(int));</pre> For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
<code>malloc ()</code> doesn't initializes the allocated memory. It contains garbage values	<code>calloc ()</code> initializes the allocated memory to zero
type cast must be done since this function returns void pointer. <pre>int*ptr; ptr = (int*)malloc(sizeof(int)*20);</pre>	Same as malloc () function <pre>int *ptr; ptr = (int*)calloc(20, 20 * sizeof(int));</pre>

Allocating a Block of Memory

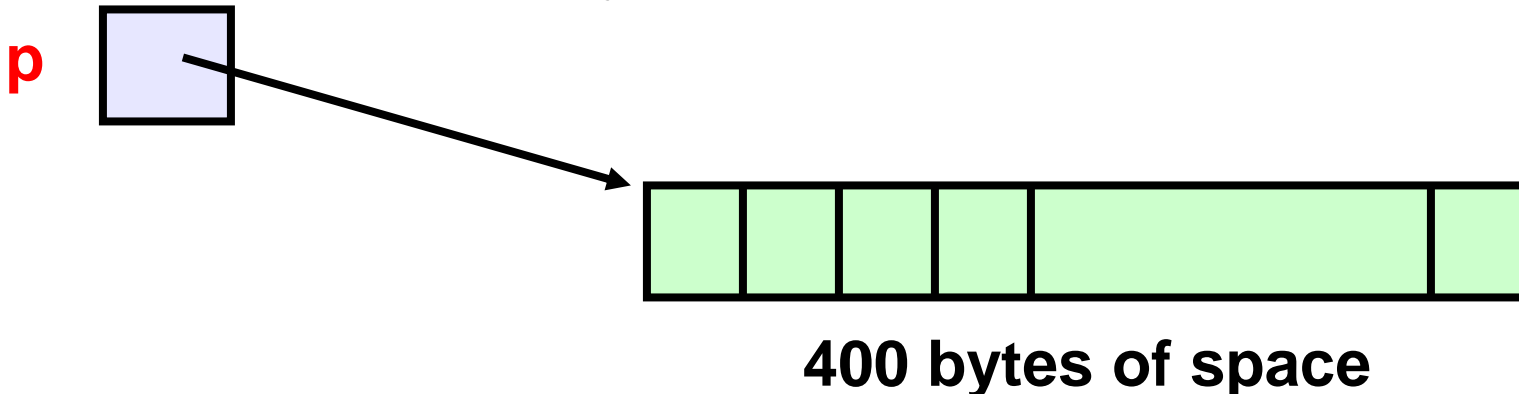
- A block of memory can be allocated using the function `malloc`
 - Reserves a block of memory of specified size and returns a pointer of type `void`
 - The return pointer can be type-casted to any pointer type
- General format:

```
type *p;  
p = (type *) malloc (byte_size);
```

Example

```
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**



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**

```
if ((p = (int *) malloc(100 * sizeof(int))) == NULL)
{
    printf ("\n Memory cannot be allocated");
    exit();
}
```

Using the malloc'd Array

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

```
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

```
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));
```

```
    printf("Input heights for %d
students \n",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);

    free (height);
    return 0;
}
```


Releasing the Allocated Space:

free

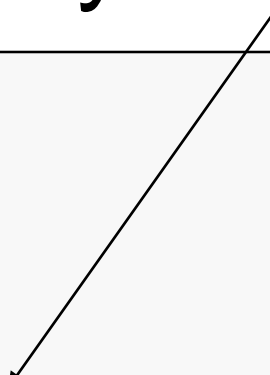
- An allocated block can be returned to the system for future use by using the **free** function
- General syntax:
free (ptr);
where **ptr** is a pointer to a memory block which has been previously created using **malloc**
- Note that no size needs to be mentioned for the allocated block, the system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

Can we allocate only arrays?

- malloc can be used to allocate memory for single variables also
 - `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
 - Array with only one element

Static array of pointers

```
#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;
}
```



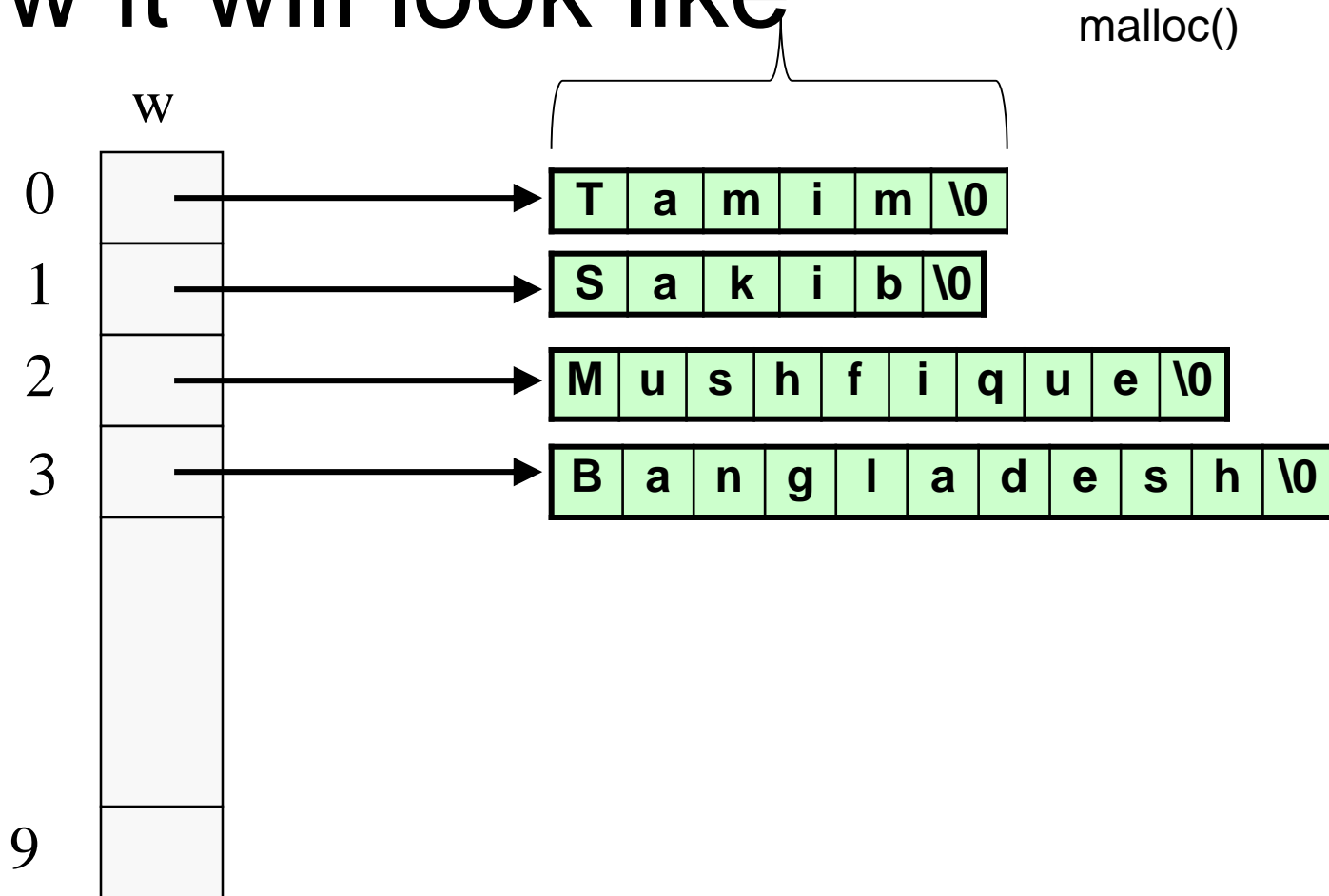
Static array of pointers

```
#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
Tamim
Sakib
Mushfique
Bangladesh
w[0] = Tamim
w[1] = Sakib
w[2] = Mushfique
w[3] = Bangladesh
```

How it will look like



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

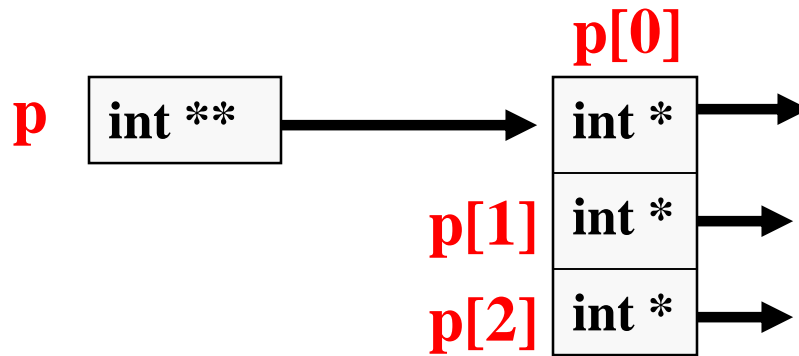
```
int x = 10, *p, **q;  
p = &x;  
q = &p;  
printf("%d %d %d", x, *p, *(*q));
```

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

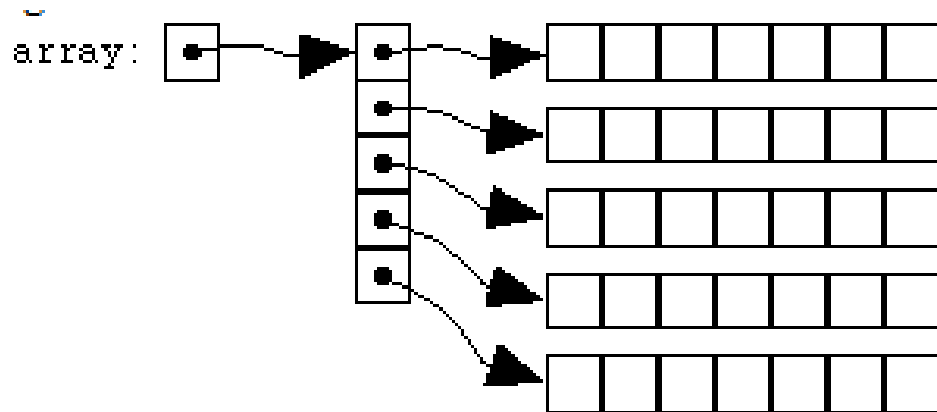
Allocating Pointer to Pointer

```
int **p;
```

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



2D array



```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
    int **array;
```

```
    array = (int**) malloc(nrows * sizeof(int *));
```

```
    for(i = 0; i < nrows; i++)
```

```
    {
```

```
        array[i] = (int*)malloc(ncolumns * sizeof(int));
```

```
    }
```

```
{
```


Dynamic Allocation of 2-d Arrays

- Recall that address of $[i][j]$ -th element is found by first finding the address of first element of i -th row, then adding j to it
- Now think of a 2-d array of dimension $[M][N]$ as M 1-d arrays, each with N elements, such that the starting address of the M arrays are contiguous (so the starting address of k -th row can be found by adding 1 to the starting address of $(k-1)$ -th row)
- This is done by allocating an array p of M pointers, the pointer $p[k]$ to store the starting address of the k -th row

Contd.

- Now, allocate the M arrays, each of N elements, with $p[k]$ holding the pointer for the k-th row array
- Now p can be subscripted and used as a 2-d array
- Address of $p[i][j] = *(p+i) + j$ (note that $*(p+i)$ is a pointer itself, and p is a pointer to a pointer)

Dynamic Allocation of 2-d Arrays

```
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);  
}
```

**Allocate array
of pointers**




**Allocate array of
integers for each
row**



```
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.**



Contd.

```
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");  
        }  
    }
```

```
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;  
}
```

Contd.

```
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");
    }
}
```

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

```
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;
}
```

Memory Layout in Dynamic Allocation

```
int main()
{
    int **p,i,j;
    int M, N;
    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 ("%10d", &p[i][j]);
        printf("\n");
    }
    return 0;
}
```

```
int **allocate (int h, int w)
{
    int **p;
    int i, j;

    p = (int **)malloc(h*sizeof (int *));
    for (i=0; i<h; i++)
    {   printf(“%10d”, &p[i]);   }
    printf(“\n\n”);
    for (i=0;i<h;i++)
    {   p[i] = (int *)malloc(w*sizeof(int));   }
    return(p);
}
```

Output

3 3

11801880 11801884 11801888

11809160 11809164 11809168

11809184 11809188 11809192

11809208 11809212 11809216

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

Elements in each row are contiguous