Data and File Structures Laboratory

Review of C – Arrays, Pointers, Dynamic Memory Allocation, Structures

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

July, 2018

What is an array?

	$A[0]$	$A[1]$	$A[2]$	...	$A[n-1]$
A	At xxxx	At xxxx+b	At xxxx+2b	...	At xxxx+(n-1)*b

What is an array?

	$A[0]$	$A[1]$	$A[2]$	...	$A[n-1]$
A	At xxxx	At xxxx+b	At xxxx+2b	...	At xxxx+(n-1)*b

- Sequence of n *contiguous* memory locations
- *Length* of the array $= n$
- *Elements* of the array can be mapped to each of the n memory locations
- Elements numbered 0 through $n-1$

What is an array?

	$A[0]$	$A[1]$	$A[2]$	...	$A[n-1]$
A	At xxxx	At xxxx+b	At xxxx+2b	...	At xxxx+(n-1)*b

- Sequence of n *contiguous* memory locations
- *Length* of the array $= n$
- *Elements* of the array can be mapped to each of the n memory locations
- Elements numbered 0 through $n-1$

```
char charArray[100], c;
charArray[i] = c; // 0 <= i <= 99
int intArray[20], i, j;
intArray[0] = i;
j = intArray[19];
```

Strings

Definition

Strings are character arrays but the end of a string is marked by the first occurrence of '\0' in the array (not the last element of the array)

Strings

Definition

Strings are character arrays but the end of a string is marked by the first occurrence of '\0' in the array (not the last element of the array)

Example:

```
char Alphabet[26];
Alphabet[0] = 'A'; Alphabet[1] = 'B'; Alphabet[2] = 'C';
Alphabet[3] = '\0'; // NOT '0'
/* Alphabet now holds the string "ABC" */
```

Strings

Definition

Strings are character arrays but the end of a string is marked by the first occurrence of '\0' in the array (not the last element of the array)

Example:

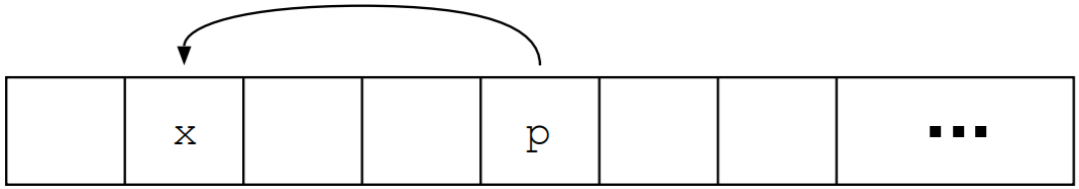
```
char Alphabet[26];
Alphabet[0] = 'A'; Alphabet[1] = 'B'; Alphabet[2] = 'C';
Alphabet[3] = '\0'; // NOT '0'
/* Alphabet now holds the string "ABC" */
```

'A'	'B'	'C'	'\0'		

The fourth cell ends the string but it is not the end of the array

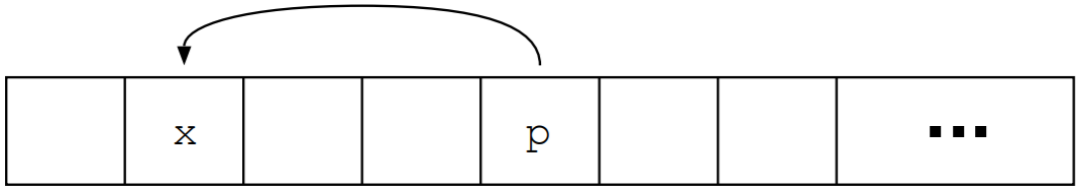
Pointers

- Memory = consecutively numbered storage cells (*bytes*)
- Variable can occupy one or more bytes
- *Address* of a variable = serial number of "first" byte occupied by the variable
- *Pointer* holds the address of a variable



Pointers

- Memory = consecutively numbered storage cells (*bytes*)
- Variable can occupy one or more bytes
- *Address* of a variable = serial number of "first" byte occupied by the variable
- *Pointer* holds the address of a variable



Note: A *pointer* variable (whatever be it pointing to) occupies the same number of bytes as that of an unsigned integer variable.

Operations with pointers

& – address / location operator (Address of ...)
* – dereferencing operator (Value at address ...)

```
int i, *ip; ip = &i;
*ip = 10; // same as i = 10

char c, *cp; cp = &c;
*cp = 'a'; // same as c = 'a'
```

Operations with pointers

& – address / location operator (Address of ...)
* – dereferencing operator (Value at address ...)

```
int i, *ip; ip = &i;
*ip = 10; // same as i = 10
```

```
char c, *cp; cp = &c;
*cp = 'a'; // same as c = 'a'
```

`ip + n` – Points to the *n*-th object after what `ip` is pointing to
`ip - n` – Points to the *n*-th object before what `ip` is pointing to

Pointers and arrays

An array name is synonymous with the address of its first element. Conversely, a pointer can be regarded as an array of elements

starting from wherever it is pointing.

Pointers and arrays

An array name is synonymous with the address of its first element. Conversely, a pointer can be regarded as an array of elements

starting from wherever it is pointing.

```
int a[10] = {...}, *p;

p = a;          // same as p = &(a[0]);
*p = 5;         // same as a[0] = 5;
p[2] = 6;       // same as a[2] = 6;
*(a+3) = 7;     // same as a[3] = 7;
```

Pointers and arrays

An array name is synonymous with the address of its first element. Conversely, a pointer can be regarded as an array of elements

starting from wherever it is pointing.

```
int a[10] = {...}, *p;

p = a;          // same as p = &(a[0]);
*p = 5;         // same as a[0] = 5;
p[2] = 6;       // same as a[2] = 6;
*(a+3) = 7;     // same as a[3] = 7;
```

Note:

`p = a; p++;` RIGHT `a = p; a++;` WRONG

The concept of base address

The following representations are basically the same

```
a[i]
i[a]
*(a+i)
*(i+a)
```

The concept of base address

The following representations are basically the same

```
a[i]
i[a]
*(a+i)
*(i+a)
```

So, the base address becomes &a[0] = &(*(a+0)) = &(*a) = *(&a) = a and the value at base address is a[0] = *(a+0) = *a.

Allocating, deallocating and reallocating memory

Syntax:

```
(type *)malloc(n*sizeof(type))  // Default garbage value
(type *)calloc(n, sizeof(type)) // Default zero value
(type *)realloc(ptr, n*sizeof(type))

free(ptr)
```

Allocating, deallocating and reallocating memory

Syntax:

```
(type *)malloc(n*sizeof(type))  // Default garbage value
(type *)calloc(n, sizeof(type)) // Default zero value
(type *)realloc(ptr, n*sizeof(type))

free(ptr)
```

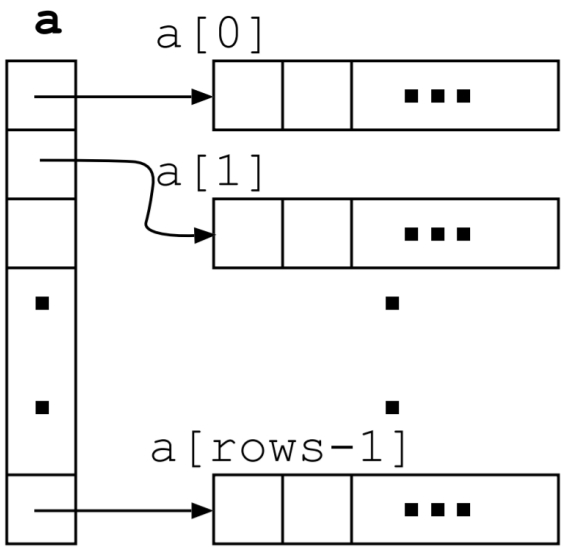
Convenient macros:

```
#define Malloc(n,type) (type *)malloc((unsigned) ((n)*sizeof(type)))
#define Realloc(loc,n,type) (type *)realloc( (char *)(loc), \
                            (unsigned) ((n)*sizeof(type)))
```

Multi-dimensional arrays

Multi-dimensional array = array of arrays = pointer to pointer

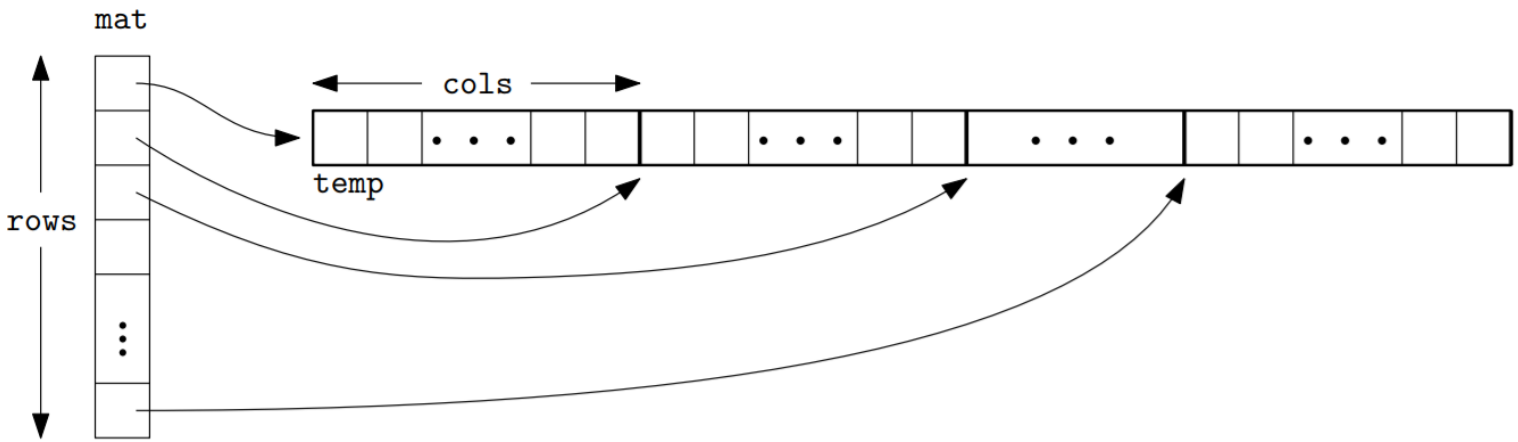
```
int **a, i;
a = (int **)malloc(rows*sizeof(int *));
for(i=0;i<rows;i++)
    a[i] = (int *)malloc(cols*sizeof(int));
```



Multi-dimensional arrays

```
for(i=0;i<rows;i++){
    free(a[i]);
}
free(a);
```

Multi-dimensional arrays




```
int ii;
int *temp;
if(NULL == (temp = (int *)malloc(rows*cols*sizeof(int))) |
    NULL == (mat = (int **)malloc(rows * sizeof(int *))))
    ERR_MESG("Out of memory");
for(ii=0;ii<rows;temp += cols,ii++)
    mat[ii] = temp;
```

Review questions

1. Suppose `s` and `t` are strings. What does the following do?
   ```
   while(*s++ = *t++);
   ```
2. What output is generated by the following code?
   ```
   for(i=0;i<=10;i++)
       printf("abcdefghijklmnop\n" + i);
   ```

Review questions — Solutions

1 String copying

```
do{
    *s = *t;
    s++; t++;
}while(*t != '\0');
```

Review questions — Solutions

1 String copying

```
do{
    *s = *t;
    s++; t++;
}while(*t != '\0');
```

```
do{
    *s++ = *t++;
}while(*t != '\0');
```

Review questions — Solutions

1 String copying

```
do{
    *s = *t;
    s++; t++;
}while(*t != '\0');
```

```
do{
    *s++ = *t++;
}while(*t != '\0');
```

```
while((*s++ = *t++) != '\0');
```

Review questions — Solutions

2. Think of the problem this way:

```
p = "abcdefghijklmnop\n";
printf(p);
```

Review questions — Solutions

2. Think of the problem this way:

```
p = "abcdefghijklmnop\n";
printf(p);
```

```
p = "abcdefghijklmnop\n";
printf(p + 2);
```

Review questions — Solutions

2. Think of the problem this way:

```
p = "abcdefghijklmnop\n";
printf(p);
```

```
p = "abcdefghijklmnop\n";
printf(p + 2);
```

```
p = "abcdefghijklmnop\n";
printf(p + i);
```

Structures

Definition

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

Example:

```
struct point{
    int x; // An integer field
    float y; // A floating point field
}p1, p2;

struct triangle{
    struct point a, b, c;
}t;
```

Operations on structures

- Assignment to members / fields
  `p1.x = 1; p1.y = 2.0; t.a.x = 0; t.a.y = 0.5;`
- Assignment / copying of structure variables
  `struct triangle t1, t2; ... ; t2 = t1;`
- Structures may be passed to functions, and returned by functions.

Note: Comparison operators (==, !=) do not work on structures.

Structure operations

- Initialization

```
struct point{
    int x; // An integer field
    float y; // A floating point field
}p1, p2;

struct triangle{
    struct point a, b, c;
}t = {{1, 1.0},
      {-1, 1.0},
       {1, -1.0}};
```

Typedefs

```
typedef unsigned int Length;
Length len, maxlen;
Length lengths[];
typedef char *String;
String p;                    // A single string
String myStrings[128]; // An array of strings
int strcmp(String, String);
p = (String) malloc(100);
```

Typedefs

```
typedef struct {
    int x;
    float y;
}POINT;
POINT p1, p2;

typedef struct {
    struct point a, b, c;
}TRIANGLE;
TRIANGLE t;
```

Pointers to structures

```
struct point *pp; // Old scheme (before typedef)
POINT *pp;        // New scheme (after typedef)

(*pp).x = 10; (*pp).y = 1.414; // Syntax 1
pp->x =10; pp->y = 1.414;      // Syntax 2
```

Memory allocation

```
TRIANGLE *tp;

tp = (TRIANGLE *)malloc(num_triangles*sizeof(TRIANGLE));
```

Problems – Day 3

1. Consider 2 sets of integers, A and B, are stored in arrays.
   1. Write a program to find the number of (possibly overlapping) occurrences of the sequence B in A.
   2. Write a program to find whether the multisets corresponding to A and B are equal.
2. Write a program to find the highest and second highest value in an array of n integers using less than $2n - 3$ comparisons.
3. Write a program to concatenate two strings. Find whether the concatenated result is a palindrome or not. Access the strings with pointers only.
4. Dynamically allocate memories to store a pair of matrices received from the user. If feasible, return their multiplication result without using any new matrix.

Problems – Day 3

5. Suppose complex numbers are stored using structure variables representing the real and imaginary parts separately. If x and y are two such variables then write logical conditions that evaluate to TRUE if and only if:
   - $x + y$ is an imaginary number without any real component.
   - x and y are complex conjugate.
   - Both x and y are real numbers.
6. Write a program that uses pointer to structure pointers to assign values to the pointed structure variables by taking inputs from the user.