Computing Laboratory

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

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

October, 2021

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?

	$A[0]$	$A[1]$	$A[2]$	$\ldots$	$A[n-1]$
A	At xxxx	At xxxx+b	At xxxx+2b	$\ldots$	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];
```

Multi-dimensional arrays

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

Multi-dimensional arrays

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

- Sequence of $m \times n$ *contiguous* memory locations
- *Elements* of the array can be mapped to each of the $m \times n$ memory locations

Multi-dimensional arrays

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

- Sequence of $m \times n$ *contiguous* memory locations
- *Elements* of the array can be mapped to each of the $m \times n$ memory locations

```
char charArray[10][20], c;
charArray[i][j] = c;        // 0 <= i <= 9, 0 <= j <= 19
int intArray[10][20], i, j;
intArray[0][0] = i;
j = intArray[9][19];
```

Multi-dimensional arrays

Providing the size is optional if a one-dimensional array is initialized while declaration takes place.

```
int charArray[] = {'a', 'b', 'c', 'd', 'e', 'f'};
int intArray[] = {1, 2, 3, 4, 5, 6};
```

Multi-dimensional arrays

Providing the size is optional if a one-dimensional array is initialized while declaration takes place.

```
int charArray[] = {'a', 'b', 'c', 'd', 'e', 'f'};
int intArray[] = {1, 2, 3, 4, 5, 6};
```

For multi-dimensional arrays, the following scenarios will work:

```
int intArray1[2][3] = {1, 2, 3, 4, 5, 6};
int intArray2[][3] = {1, 2, 3, 4, 5, 6};
```

However, the following will not work:

```
int intArray3[2][] = {1, 2, 3, 4, 5, 6};
int intArray4[][] = {1, 2, 3, 4, 5, 6};
```

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:

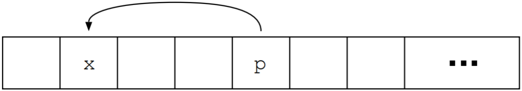
```
char Alphabet[26];
Alphabet[0] = 'A'; Alphabet[1] = 'B'; Alphabet[2] = 'C';
Alphabet[3] = '\0'; // NOT '0'
/* Alphabet now holds the string "ABC" */
```

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

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

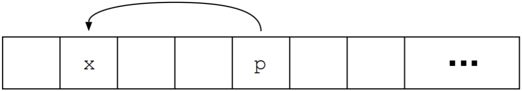
Pointers

- 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

Note: A pointer that has not yet been initialized (points to nothing) is known as a *wild* pointer.

`const char *` vs `char *const` vs `const char * const`

- `const char *ptr`: This is a pointer to a constant character. One cannot change the value pointed by ptr, but you can change the pointer itself.
- `char *const ptr`: This is a constant pointer to non-constant character. One cannot change the pointer p, but can change the value pointed by ptr.
- `const char *const ptr`: This is a constant pointer to constant character. One can neither change the value pointed by ptr nor the pointer ptr.

const char * vs char *const vs const char * const

Use of `const char *ptr`:

```
char x ='A', y ='B';
const char *ptr = &x;
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
ptr = &y; // Writing *ptr = y is illegal
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
```

const char * vs char *const vs const char * const

Use of `const char *ptr`:

```
char x ='A', y ='B';
const char *ptr = &x;
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
ptr = &y; // Writing *ptr = y is illegal
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
```

Output:

ptr is pointing to the value: A
ptr is pointing to the address: 221652998
ptr is pointing to the value: B
ptr is pointing to the address: 221652999

const char * vs char *const vs const char * const

Use of `char *const ptr`:

```
char x ='A', y ='B';
char *const ptr = &x;
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
*ptr = y; // Writing ptr = &y is illegal
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
```

const char * vs char *const vs const char * const

Use of `char *const ptr`:

```
char x ='A', y ='B';
char *const ptr = &x;
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
*ptr = y; // Writing ptr = &y is illegal
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
```

Output:

ptr is pointing to the value: A
ptr is pointing to the address: 221652998
ptr is pointing to the value: B
ptr is pointing to the address: 221652998

const char * vs char *const vs const char * const

Use of `const char *const ptr`:

```
char x ='A', y ='B';
const char *const ptr = &x;
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
// Writing ptr = &y and *ptr = y are both illegal
```

const char * vs char *const vs const char * const

Use of `const char *const ptr`:

```
char x ='A', y ='B';
const char *const ptr = &x;
printf("ptr is pointing to the value: %c\n", *ptr);
printf("ptr is pointing to the address: %u\n", ptr);
// Writing ptr = &y and *ptr = y are both illegal
```

Output:

ptr is pointing to the value: A

ptr is pointing to the address: 221652998

Void pointers

A void pointer, irrespective of the data type, points to some arbitrary data location in storage.

```
int i = 10;
char c = ‘a’;
void *p;
p = &i; // Points to an integer variable
printf("Integer value: %d", *((int*)p));
p = &c; // Points to a character variable
printf("\nCharacter value: %c", *((char*)p));
```

Void pointers

A void pointer, irrespective of the data type, points to some arbitrary data location in storage.

```
int i = 10;
char c = 'a';
void *p;
p = &i; // Points to an integer variable
printf("Integer value: %d", *((int*)p));
p = &c; // Points to a character variable
printf("\nCharacter value: %c", *((char*)p));
```

Note: A void pointer variable cannot be dereferenced in general. However, it can be done via typecasting.

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)))
```

Dangling pointers

A pointer turns out to be *dangling* if the block of memory it points to is made free.

```
int *a, cols;
a = (int *)malloc(cols*sizeof(int));
...
free(a); // Turns ‘a’ into a dangling pointer
a = NULL; // Restricts ‘a’ from being dangling
```

Dangling pointers

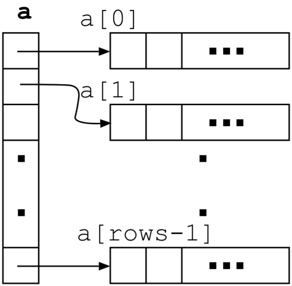
A pointer turns out to be *dangling* if the block of memory it points to is made free.

```
int *a, cols;
a = (int *)malloc(cols*sizeof(int));
...
free(a); // Turns ‘a’ into a dangling pointer
a = NULL; // Restricts ‘a’ from being dangling
```

Note: A pointer is converted to a NULL Pointer, which points to nothing, so that it no more remains dangling.

Multi-dimensional arrays

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



Multi-dimensional arrays

The memory for a two-dimensional array is dynamically allocated as follows.

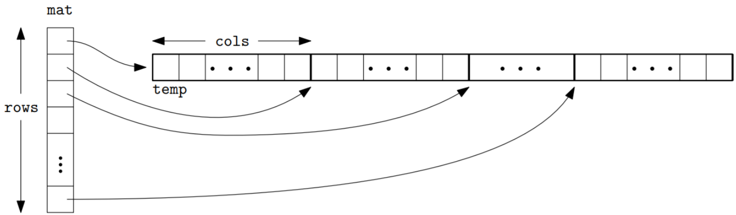
```
int **a, i, rows, cols;
a = (int **)malloc(rows*sizeof(int *));
for(i=0;i<rows;i++){
    a[i] = (int *)malloc(cols*sizeof(int));
}
```

Multi-dimensional arrays

The dynamically allocated memory for a two-dimensional array is freed as follows.

```
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.

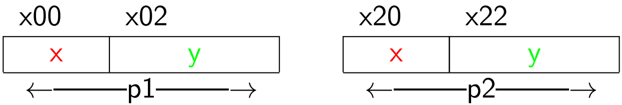
Example:

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

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

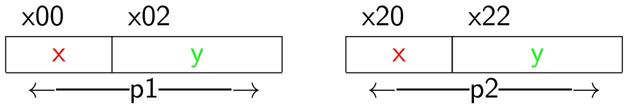
Structures

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

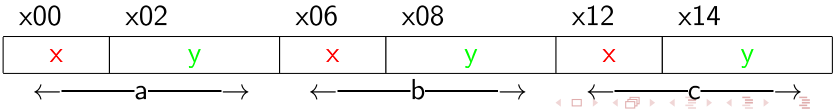


Structures

```
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 initialization

- 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 UI;
UI i, j;                // A pair of unsigned integers
UI myArray[10];         // An array of unsigned integers

typedef char *STR;
STR p;                  // A single string
STR myStrings[128];     // An array of strings

p = (STR) malloc(100);
UI strcmp(STR, STR); // Prototype declaration
```

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. Two elements $A[i]$ and $A[j]$ of an array A are said to form an *inversion pair* if $A[i] > A[j]$ but $i < j$. Write a program to count the number of inversion pairs in a list A containing distinct integers.
   Note that, for the array $A = \{8, 4, 2, 1\}$, the *inversion pairs* are (8, 4), (4, 2), (8, 2), (8, 1), (4, 1) and (2, 1).
2. Dynamically allocate memories to store a matrix received from the user. If the matrix is not *invertible* return `SINGULAR`. Otherwise, return its inverse matrix without using any new matrix.
3. Let there be a pair of sorted arrays, A and B, and an index k such that $0 \leq k < |A| + |B|)$ given as user inputs. Find the k^{th} smallest element in $A \cup B$ without merging the two arrays.

Problems – Day 3

4. 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.
   - $x - y$ is a real number without any imaginary component.
   - x and y are complex conjugate.
   - Both x and y are real numbers.
5. Write a program that uses pointer to structure pointers to assign values to the pointed structure variables by taking inputs from the user. You are free to choose your own problem to use the said constructs.