COMsW 1003-1

Introduction to Computer Programming in C

Lecture 12

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

Homework 3 is out

- Due on Monday, 03/21/11 at the beginning of class, no exceptions

Midterm

- In class on Wednesday, 03/09/11
- Will cover everything up to Lecture 13 (included)
- Open books, open notes
- Closed electronic devices

Today

- Passing arguments to function by value vs. by reference (from Lec 11)
- Functions returning pointers
- Pointers of pointers

Functions Returning Pointers

- Naturally, a function can return a pointer
- This is a way to return an array, but must be **careful** about what has been **allocated** in memory

```
returnType * functionName( parmeters )
```

NOTE

`NULL` is the equivalent of zero for pointers splitString.c

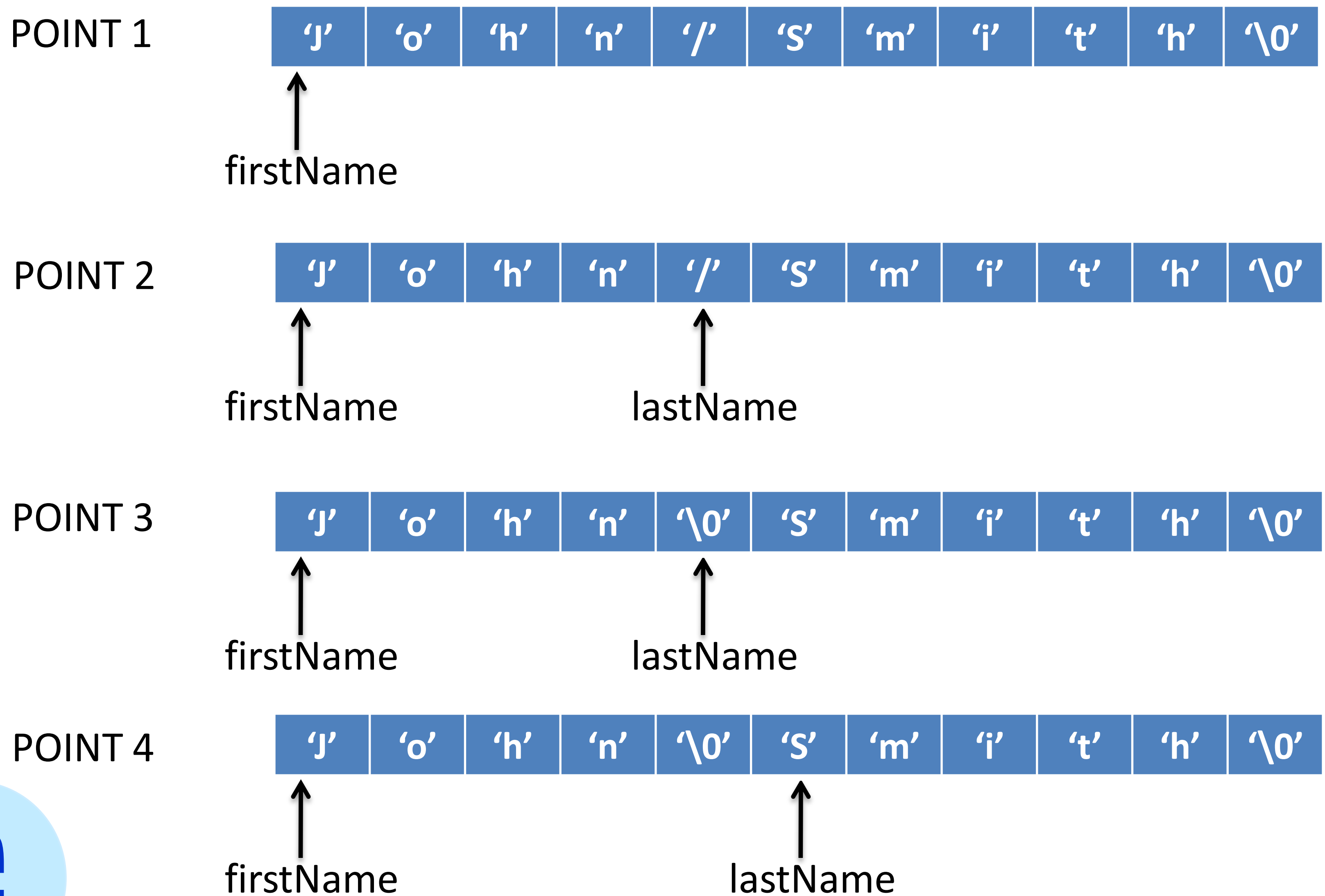
Functions Returning Pointers

Example: using pointers to return a string

Given a string of the type “firstNAme/lastName”
We want to split it into two separate entities to print

splitString.c

Functions Returning Pointers



Const pointers

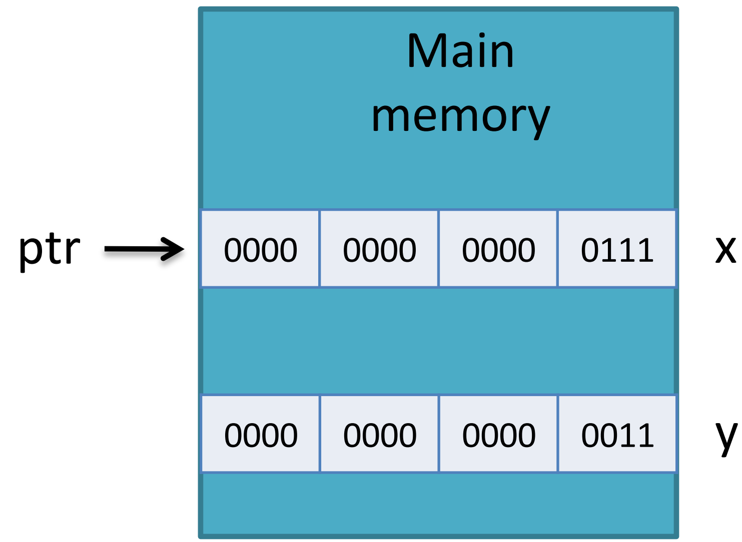
point.c

```
const type *
```

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;
```



Const pointers

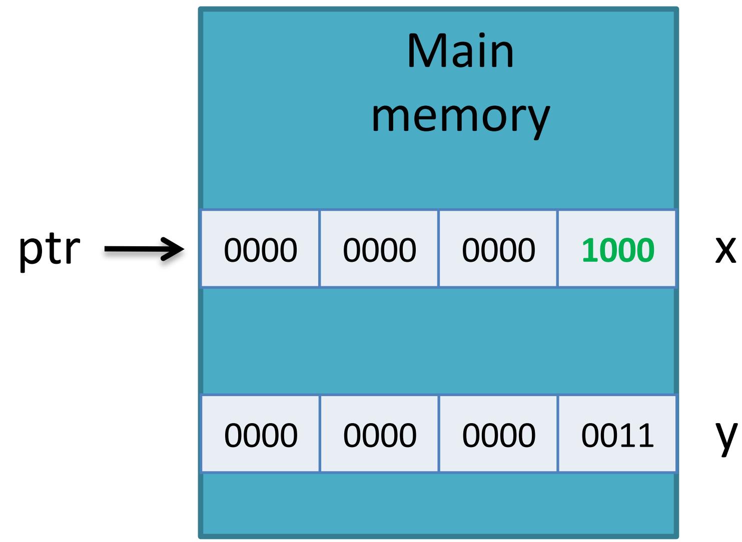
point.c

```
const type *
```

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;

x = 8;
```



Const pointers

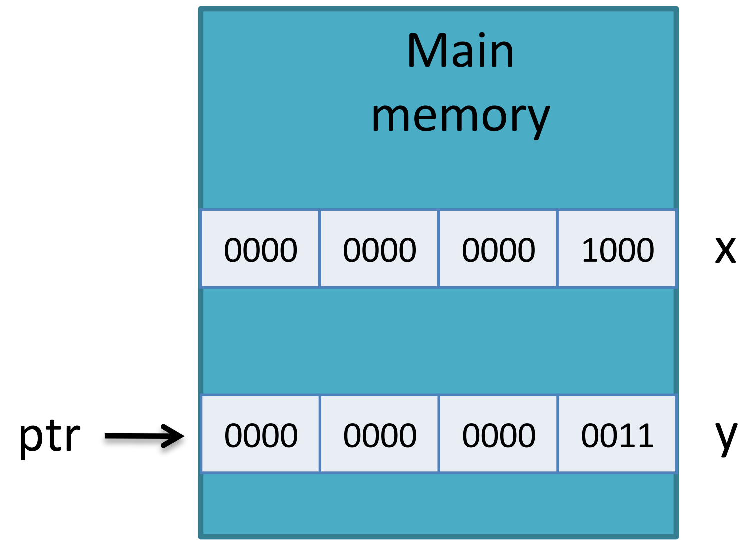
point.c

```
const type *
```

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;   ✗

x = 8;   V

ptr = &y;   V
```



Const pointers

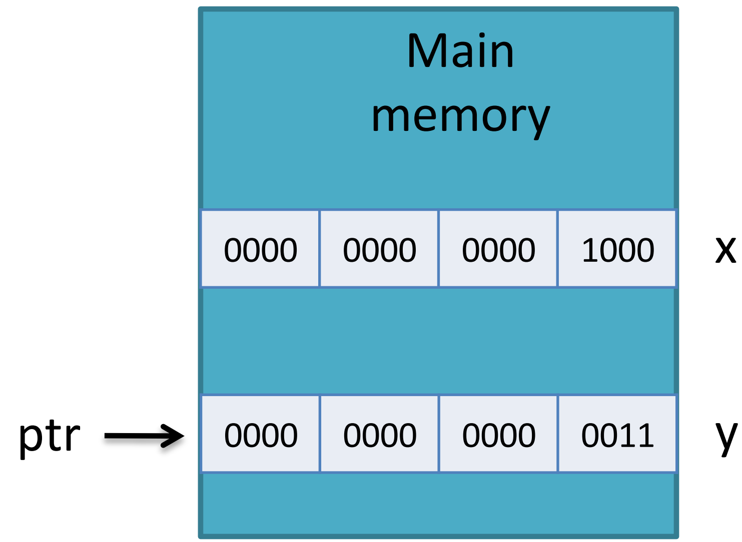
point.c

```
const type *
```

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;   ✗

x = 8;   V

ptr = &y;   V

*ptr = 9;   ✗

printf("x = %d, y = %d\n",x,*ptr);
```



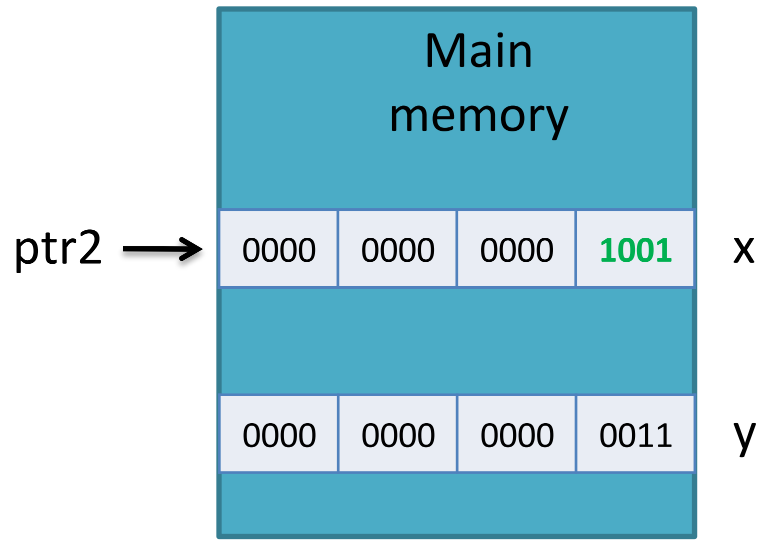
Const pointers

point.c

```
type * const
```

This is the declaration of a constant pointer. In this case, the pointer is fixed, but the value at the address it points to can be modified

```
int x = 7, y = 3;

int * const ptr2 = &x;

*ptr2 = 9;

ptr2++;

ptr2 = &y;


printf("x = %d, x = %d\n", x, *ptr2);
```



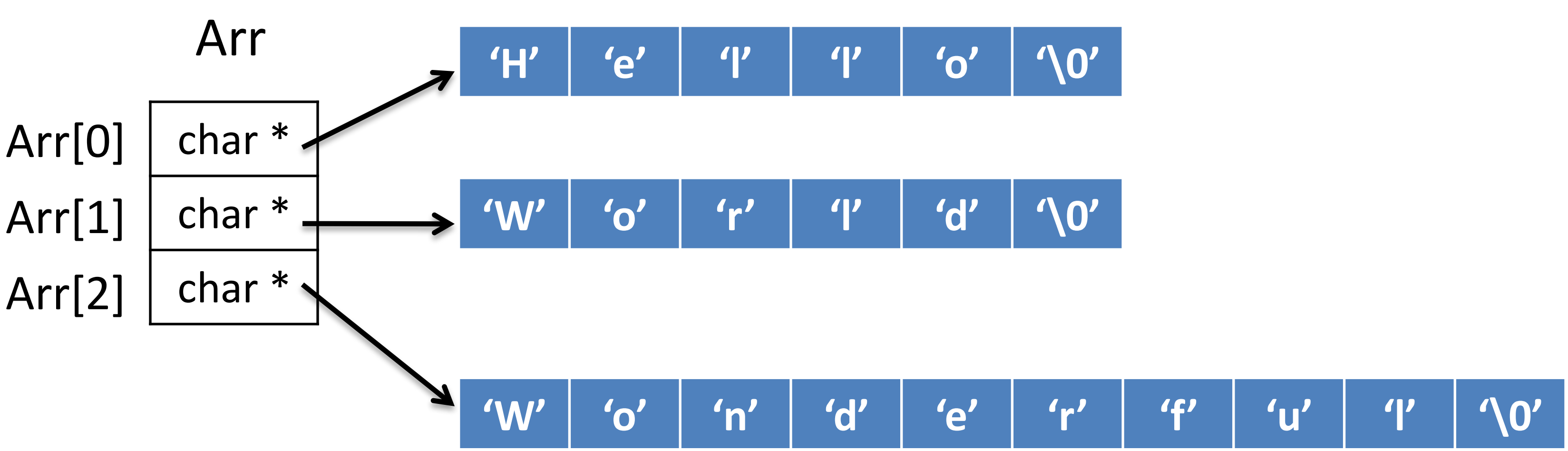
stringArrays.c

Arrays of strings

- An array Arr of 3 strings of variable length

```
char *Arr[3]={ "Hello", "World", "Wonderful" };

Arr[2] ⟷ Arr+2 // "Wondeful"
```

- Arr is an array of **3** elements. Each element in Arr is of type **pointer to char**.



stringArrays.c

Arrays of strings

- An array Arr of 3 strings of variable length

```
char *Arr[3]={ "Hello", "World", "Wonderful" };

Arr[2] ⟷ Arr+2 // "Wondeful"
```

- An array Arr of 3 strings of maximum length = 15

```
char Arr2[3][15] = { "Hello2", "World2", "Wonderful2" };

Arr2[0]⟷ Arr2 // "Hello2"

Arr2[1] ⟷ Arr2+1 // "World2"
```

stringArrays.c

Pointers of pointers

Arr

	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

Arr2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘2’	‘\0’								
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘2’	‘\0’								
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘2’	‘\0’				

stringArrays.c

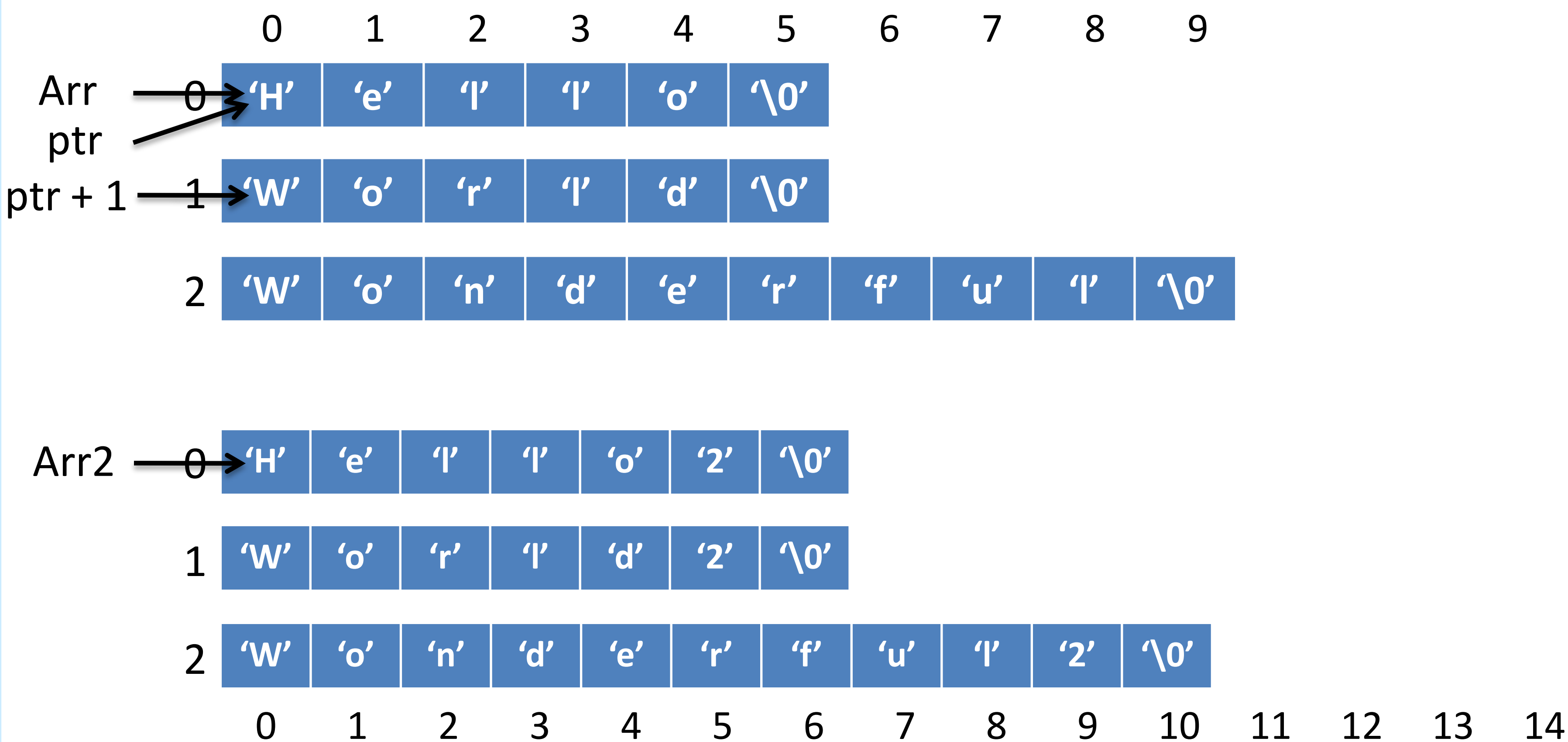
Pointers of pointers

- A pointer can point to another pointer
- In a sense, it's the equivalent of matrices!

```
int x = 3;

int *p = &x;

int **p2 = &p;

x = 2;   ⟷   *p = 2;   ⟷   **p2 = 2;


char *Arr[3]={ "Hello", "World", "Wonderful" };

char **ptr;

ptr = Arr;
```

stringArrays.c

Pointers of pointers



Pointers of pointers

stringArrays.c

```
char *Arr[3]={ "Hello", "World", "Wonderful" };
char **ptr;
ptr = Arr;
```

((ptr+1)+2) ?

Pointers of pointers

stringArrays.c

```
char *Arr[3]={ "Hello", "World", "Wonderful" };
char **ptr;
ptr = Arr;
```

((ptr+1)+2)

1 . ptr+1

	0	1	2	3	4	5	6	7	8	9
0	'H'	'e'	'l'	'l'	'o'	'\0'				
1	'W'	'o'	'r'	'l'	'd'	'\0'				
2	'W'	'o'	'n'	'd'	'e'	'r'	'f'	'u'	'l'	'\0'

ptr + 1 → 1

ptr+1 points to the whole line

2 . *(ptr+1)

	0	1	2	3	4	5	6	7	8	9
0	'H'	'e'	'l'	'l'	'o'	'\0'				
1	'W'	'o'	'r'	'l'	'd'	'\0'				
2	'W'	'o'	'n'	'd'	'e'	'r'	'f'	'u'	'l'	'\0'

*(ptr + 1) → 1

*(ptr+1) points to the first element of the line

Pointers of pointers

stringArrays.c

```
char *Arr[3]={ "Hello", "World", "Wonderful" };
char **ptr;
ptr = Arr;
```

((ptr+1)+2)

3 . *(ptr+1)+2

*(ptr+1)+2 points to the third element of the line

*(ptr + 1)+2 →

	0	1	2	3	4	5	6	7	8	9
0	'H'	'e'	'l'	'l'	'o'	'\0'				
1	'W'	'o'	'r'	'l'	'd'	'\0'				
2	'W'	'o'	'n'	'd'	'e'	'r'	'f'	'u'	'l'	'\0'

2 . *(*(ptr+1)+2)

Now we get the value stored at the address we pint

((ptr+1)+2) →

	0	1	2	3	4	5	6	7	8	9
0	'H'	'e'	'l'	'l'	'o'	'\0'				
1	'W'	'o'	'r'	'l'	'd'	'\0'				
2	'W'	'o'	'n'	'd'	'e'	'r'	'f'	'u'	'l'	'\0'

Pointers of pointers

stringArrays.c

```
char *Arr[3]={ "Hello", "World", "Wonderful" };
char **ptr;
ptr = Arr;
```

((ptr+1)+2) ←

Avoid this notation!
ptr[1][2] is much better!

	0	1	2	3	4	5	6	7	8	9
0	'H'	'e'	'l'	'l'	'o'	'\0'				
1	'W'	'o'	'r'	'l'	'd'	'\0'				
2	'W'	'o'	'n'	'd'	'e'	'r'	'f'	'u'	'l'	'\0'

Pointers vs. Arrays

	Arrays		Pointers
1D array of 5 int	`int x[5];`	⟷	`int *xPtr;`
2D array of 6 int 2x3 matrix	`int y[2][3];`	⟷	`int **yPtr;`
2D array of 4 int 2x2 matrix	`int* z[2]={{1,2},{2,1}};`	⟷	`int **zPtr;`
1D array of 5 char string	`char c[] = “mike”;`	⟷	`char *cPtr;`

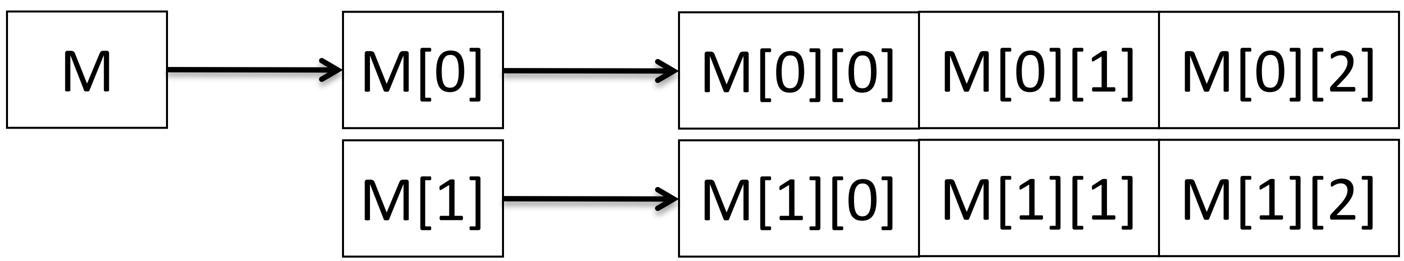
Space has been allocated in memory for the arrays

Space has been allocated in memory only for the pointers variables, **NOT** for the arrays they will point to.
The DIMENSIONS of the arrays are UNKNOWN

Multidimensional Arrays

2x3 matrix of double

```
double M0[2][3];
double *M1[2] = M0;
double **M = M0;
```



double ** double * double

Multidimensional Arrays

2x3 matrix of double

```
double M0[2][3];

double *M1[2] = M0;

double **M = M0;
```

The difference between M0, M1 and M is that
M1 and M can have ANY SIZE !