COMsW 1003-1

Introduction to Computer Programming in C

Lecture 15

Spring 2011

Instructor: Michele Merler

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

Announcements

Homework 4 out, due April 11th at the beginning of class

Read CPL Chapter 5

Today

- Finish C Standard Libraries
- Pointers to void
- Begin Dynamic Memory Allocation

Review : operators * and &

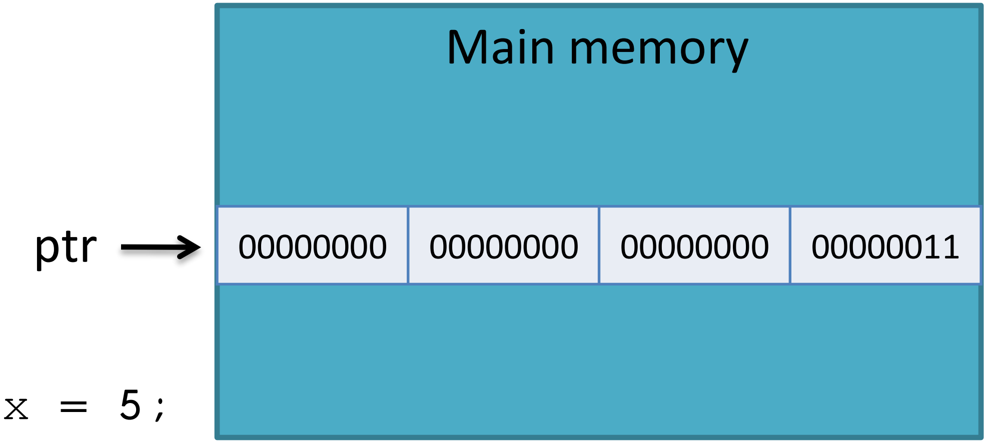
*** dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```
int x = 3;

int *ptr;

ptr = &x;

*ptr = 5; // x = 5;
```

Make ptr point to the address of x

Modify the value in address pointed by ptr



Main memory

ptr

00000000	00000000	00000000	00000011

Review : 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;
```

Review: 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;`

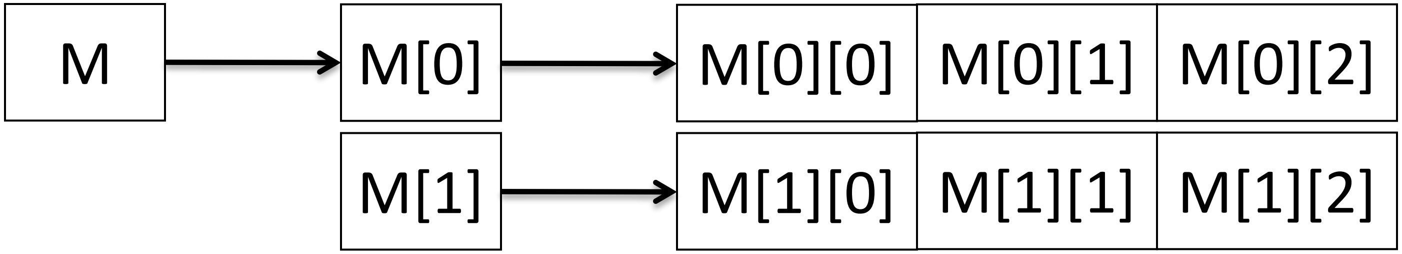
Space has been allocated in memory for the arrays

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

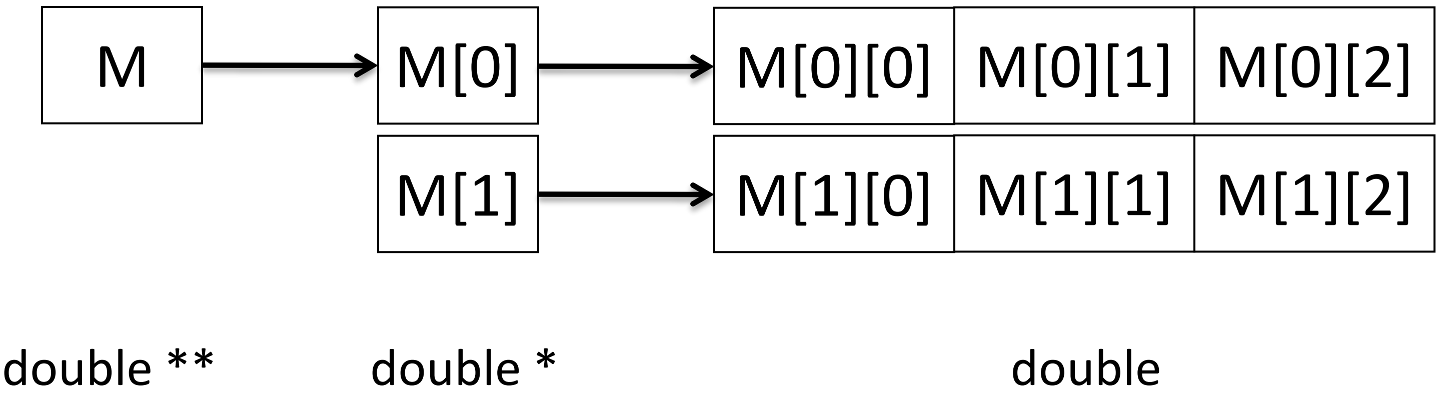
Multidimensional Arrays

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 !



M

M[0]

M[1]

M[0][0] M[0][1] M[0][2]

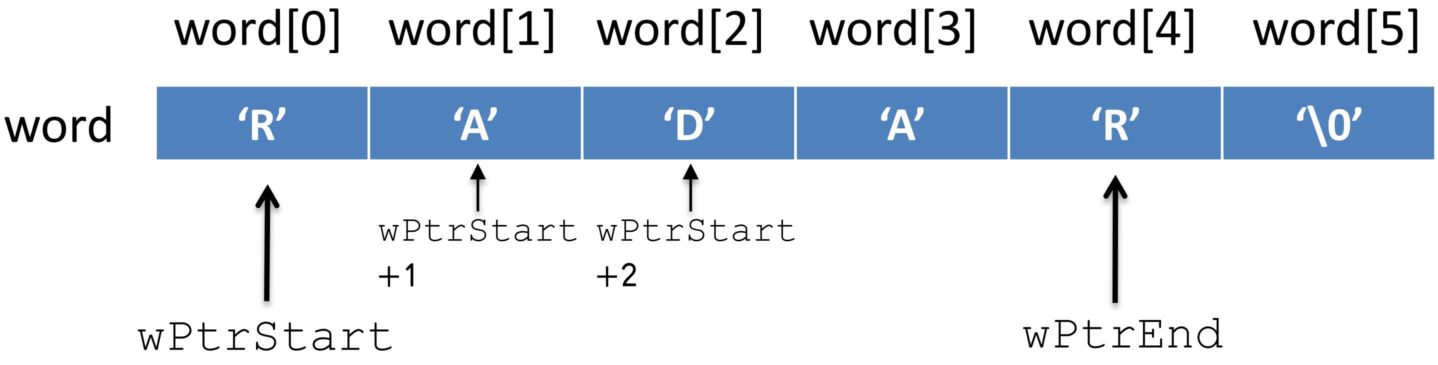
M[1][0] M[1][1] M[1][2]

double **

double *

double

Review : Pointers and Arrays



	word[0]	word[1]	word[2]	word[3]	word[4]	word[5]
word	‘R’	‘A’	‘D’	‘A’	‘R’	‘\0’

wPtrStart

wPtrStart +1

wPtrStart +2

wPtrEnd

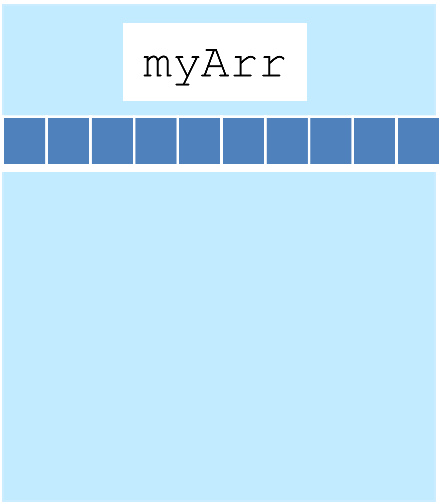
```
char word[8] = “RADAR”;

char *wPtrStart = word;
```

`char*` is a string

Pointers vs. Arrays

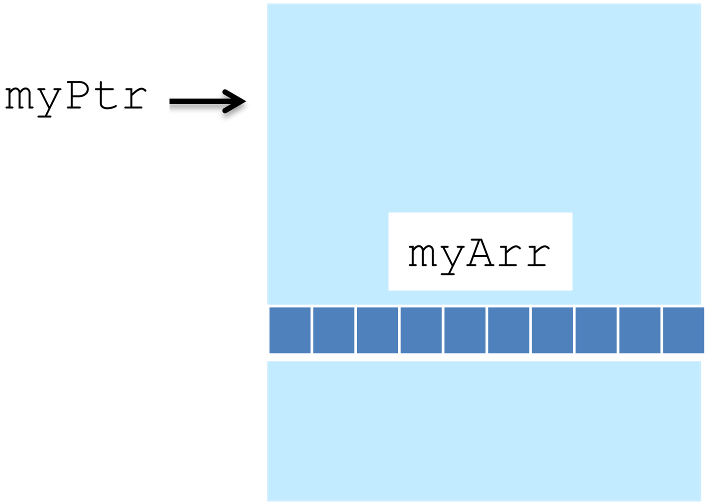
- Arrays represent actual memory **allocated** space

```
int myArr[10];
```



- Pointers **point** to a place in memory

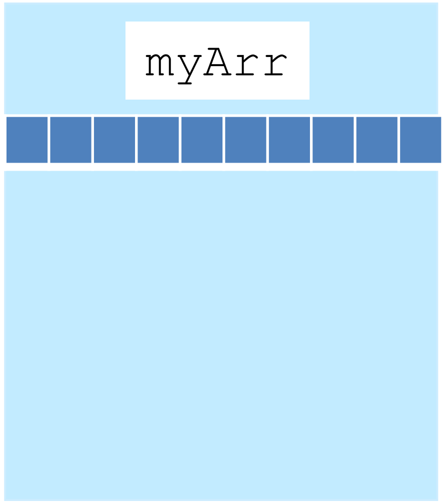
```
int *myPtr;
```



Pointers vs. Arrays

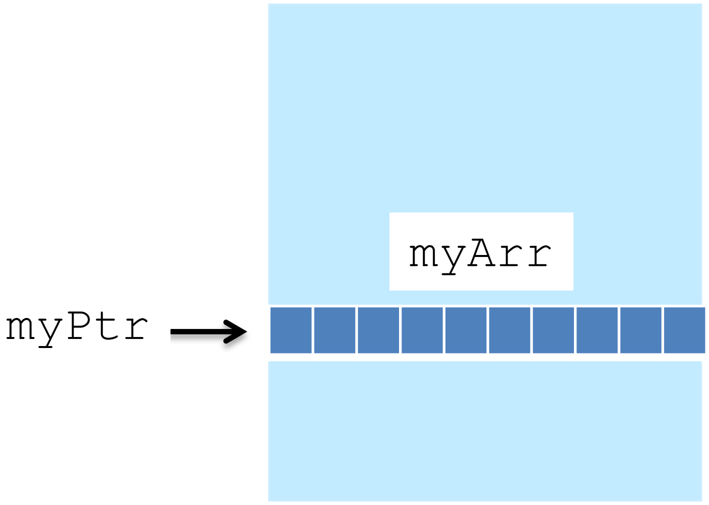
- Arrays represent actual memory **allocated** space

```
int myArr[10];
```



- Pointers **point** to a place in memory

```
int *myPtr;
myPtr = myArr;
```



sizeof()

- So far, we have been using sizeof() to determine the length of a string (including ‘\0’)

- sizeof() is a more general function, that returns the size, measured in bytes, of a variable or a type

```
size_t sizeof( var )
```

- `size_t` can be used (implicitly casted) as an integer voidFun.c

Void *

`void *` means a pointer of ANY type

Sometimes functions can use `void *` as argument and return type.

This allows the programmer to specify the type of pointer to use at **invocation time**

This is a form of function overloading (popular in C++)

```
void *function_name( void *arg1, … , void *argN )
```

```
int i;
double d;
int *pi;
double *pd;

void *pv;
```

Void *

```
pi = &d;        // Compiler warning
pd = &i;        // Compiler warning

pv = &i;        // OK
printf(”%d\n”, *pv);  // Compiler error
printf(”%d\n”, *(int *)pv);    // OK

pv = &d;                 // OK
printf(”%f\n”, *pv);  // Compiler error
printf(”%f\n”, *(double *)pv); // OK

pv = &i;               // OK
d = *(double *)pv;  // Runtime error
```

Void *

voidFun.c

Example

```c
void *pointElement( void *A, int ind, int type ){

   if( type == 1 ){
       return( A + sizeof(int) * ind  );
   }
}

int main(){

  int M[3] = {1 , 2, 3};
  int element = 1;

  int *M2 = (int *) pointElement( M , element, 1);

}
```

voidFun.c

Void *

Example

```
void *pointElement( void *A, int ind, int type ){

   if( type == 1 ){
       return( A + sizeof(int) * ind  );
   }
}

int main(){

  int M[3] = {1 , 2, 3};
  int element = 1;

  int *M2 = (int *) pointElement( M , element, 1);

}
```

Explicit cast

Dynamic Memory Allocation

Functions related to DMA are in the library **stdlib.h**

```
void *malloc( size_t numBytes )
```

Allocates *numBytes* bytes in memory (specificaly, in a part of memory called heap)

The elements in the allocated memory are not initialized

Returns a pointer to the allocated memory on success, or NULL on failure

```
void *calloc( size_t numElements,  size_t size )
```

Allocates `size*numElements` bytes in memory

All elements in the allocated memory are set to zero

Returns a pointer to the allocated memory on success, or NULL on failure

Dynamic Memory Allocation

Example: create an array of 10 integers

```
int myArr[10];
```

- Malloc()

  Example

  ```
  int *myArr = (int *) malloc( 10 * sizeof(int) );
  ```

- Calloc()

  Example

  ```
  int *myArr = (int *) calloc( 10 , sizeof(int) );
  ```

Dynamic Memory Allocation

Functions related to DMA are in the library **stdlib.h**

```
void *realloc(void *ptr, size_t size)
```

Changes the size of the allocated memory block pointed by *ptr* to *size*

Returns a pointer to the allocated memory on success, or NULL on failure

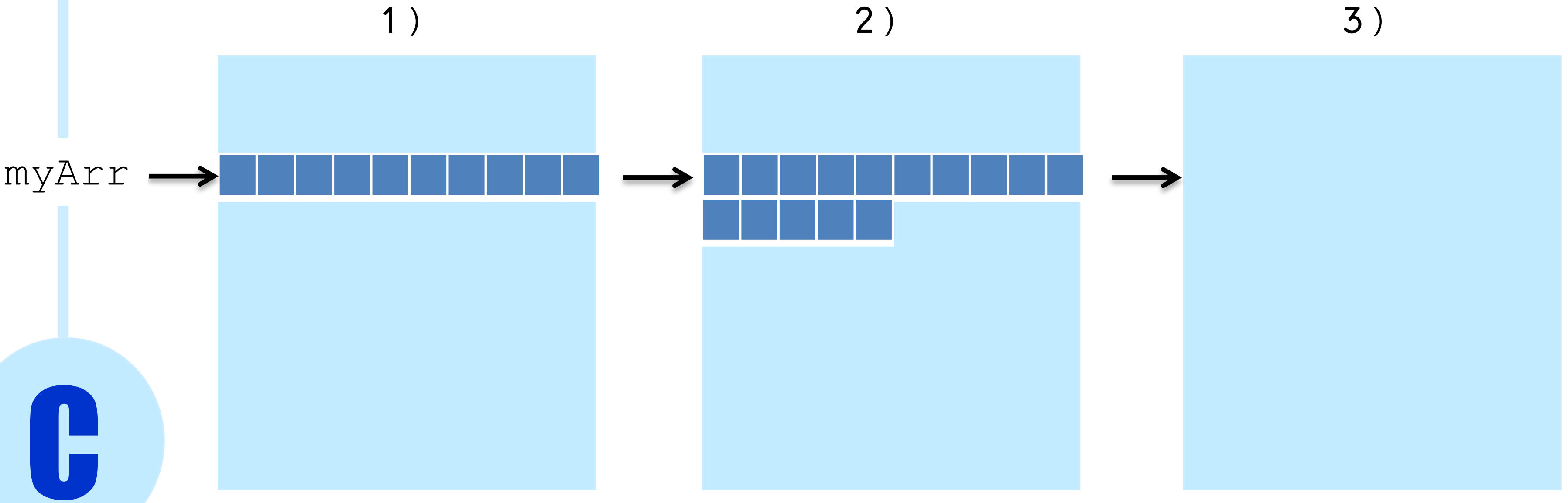
```
void free(void *ptr)
```

De-allocates (frees) the space in memory pointed by `ptr`

Dynamic Memory Allocation

Example: create an array of 10 integers, resize it to 15, then free the space in memory

```
1) int *myArr = (int *) malloc( 10 * sizeof(int) );

2) myArr = realloc( myArr, 15 * sizeof(int) );

3) free( myArr );
```



commandLine.c

Dynamic Memory Allocation

Example: reading an indefinitely long command line

So far we have been reading strings from command line using an array

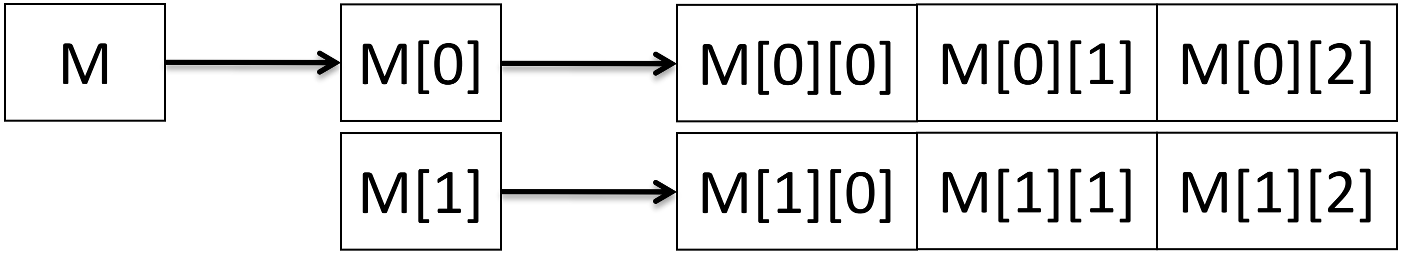
```
char line[100];
fgets( line, sizeof(line), stdin);
```

What if the user enters a command with 105 characters?

Dynamic Memory Allocation

Multidimensional Arrays

2x3 matrix of double



M

M[0]

M[1]

M[0][0] M[0][1] M[0][2]

M[1][0] M[1][1] M[1][2]

double ** double * double

Dynamic Memory Allocation

Multidimensional Arrays

2x3 matrix of double

```
double** M = (double**) malloc( 2 * sizeof(double *) );

int i;
for ( i = 0 ; i<2; i++ ){
    M[i] = malloc( 3 * sizeof(int) );
}

/* use M as a regular 2-dimensional array */

for ( i = 0 ; i<2; i++ ){
    free( M[i] );
}
free( M );
```

Memory Leaks

Space in the heap is LIMITED, therefore we must be careful and free memory

There are two cases in whish freeing memory becomes impossible:

- when we move a pointer after allocating memory

```
int N = 40000;

char *str = “Hello”;

char *giantString = malloc(N*sizeof(char));

giantString = str;
```

Now we cannot find anymore the location of the block of allocated memory

Memory Leaks

Space in the heap is LIMITED, therefore we must be careful and free memory

There are two cases in whish freeing memory becomes impossible:

- if we reallocate memory using the same pointer

```
int N = 40000;

char *giantString = malloc(N*sizeof(char));

/* do something */

giantString = malloc(N*sizeof(char));
```

giantString now points to a newly allocated block of memory, the location of the previous one is lost