# CS 261 – Data Structures

Introduction to

C Programming

# Why C?

- C is a lower level, imperative language
- C makes it easier to focus on important concepts for this class, including
  - **memory allocation**
  - **execution time complexity**
- Lays groundwork for many other languages

# Files: Interface and Implementation

- Interface files (***.h**)
- Implementation files (***.c**)

```
#include <stdio.h>
#include <stdlib.h>
#include "my_file.h"
int main (int argc, char *argv[]) {
    /*your code*/ }
void my_function(double a) {
    /*your code*/ }
```

  - Every code must have **main()**
  - **main()** does not need to contain the return statement

# Interface File

- Interface files (`*.h`) have
  - Declarations of variables
  - Declarations of types,
  - Preprocessor commands,
  - Function prototypes -- header but no body:
    - Example: `int max(int a, int b);`

      terminated with a semicolon!

# Declarations of Variables

- When you declare a variable, a memory space is reserved for that variable

```
int i;    /* 8 bytes for 64-bit machine */
double  d;
long test[100];/* reserved 100 locations of size long */
```

- Note that the index for the above array goes from 0 to 99
  `test[100] = 4;` /* ERROR !!!*/

# Declarations of Types and Constants

- Examples:

```
/* constant TYPE is declared as type double */
# define  TYPE  double
# define  TYPE  char

/* Replaces MAX in  code with 423 */
# define MAX 423
```

# Function Definitions

```
returnType  functionName(arguments)  {
    declarations of variables;/*Must come first*/
    commands;
}
```

# Function Definitions -- Example

Return a sum of n integers:

```
long arrSum(int arr[], unsigned int n)
 {
   /*unsigned int i;/* Loop variable. */*/
   long sum = 0;   /* Sum initialized to zero. */

   for (int i = 0; i < n; i=i+1) {
     sum = sum + arr[i];
   }
   return sum;
 }
```

Need to pass size of array (not included in **`arr`**).

# Variable and its Memory Location

```
double mass;  /* variable */
long memory; /* variable */

mass = 0.01;

memory = & mass;

printf("%e,   %p \n",mass,memory);
```

Output: `1e-2,   ffbff958`

# Pointers

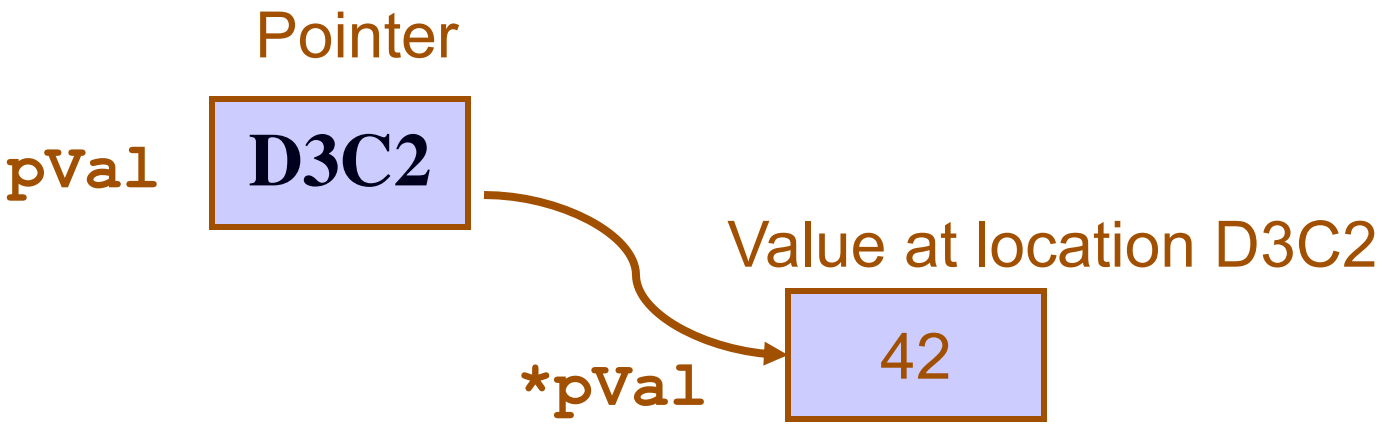
- A pointer is a variable that refers to a memory location

# Pointer Value vs. Thing Pointed At

the value of the pointer

vs.

the value of the thing the pointer points to:



# Pointer Syntax

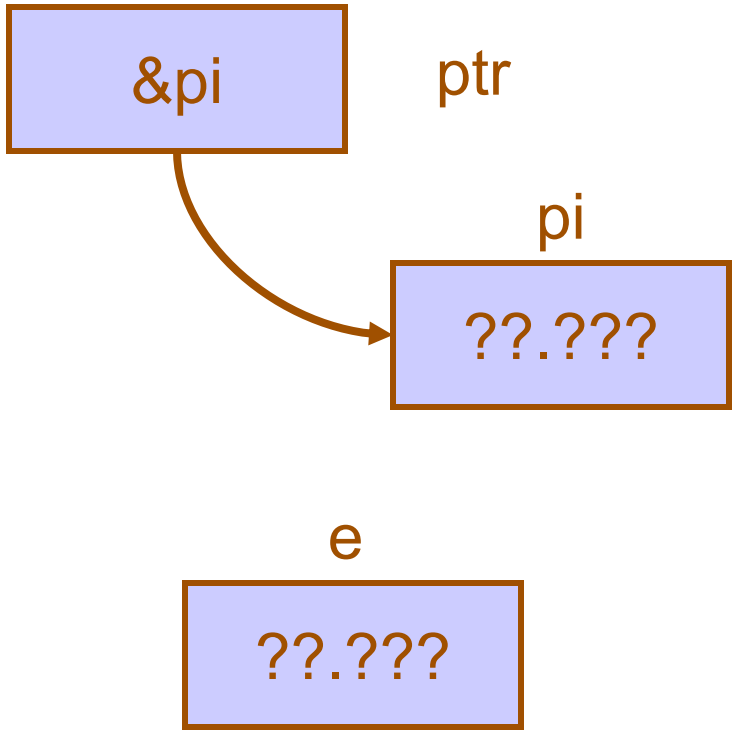
- Use `*` to
  - declare a pointer,
  - get value of pointer

- Use `&` to get address of a variable

```
double *ptr;
double pi, e;
```

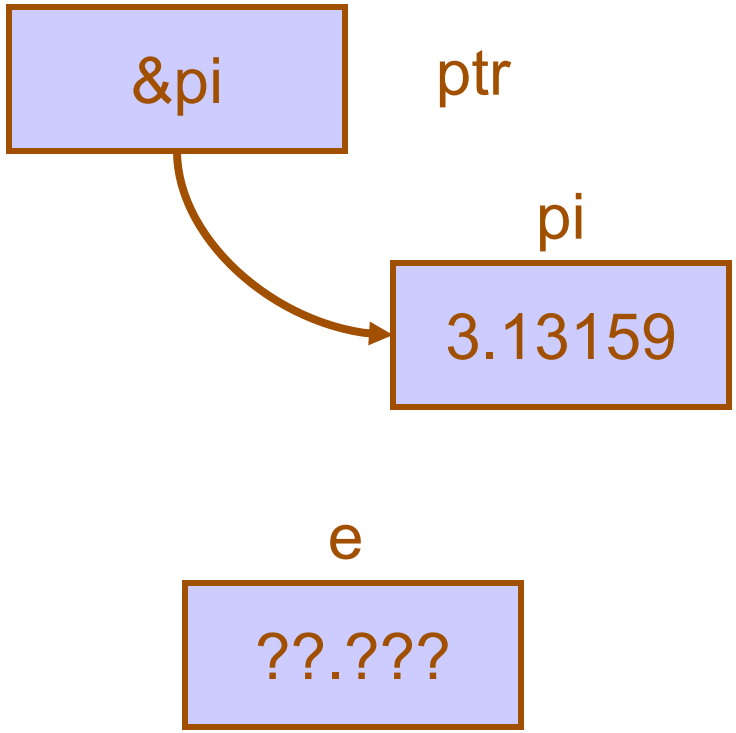
# Pointer Syntax -- Example

```
double *ptr;
double pi, e;

ptr = &pi;
```



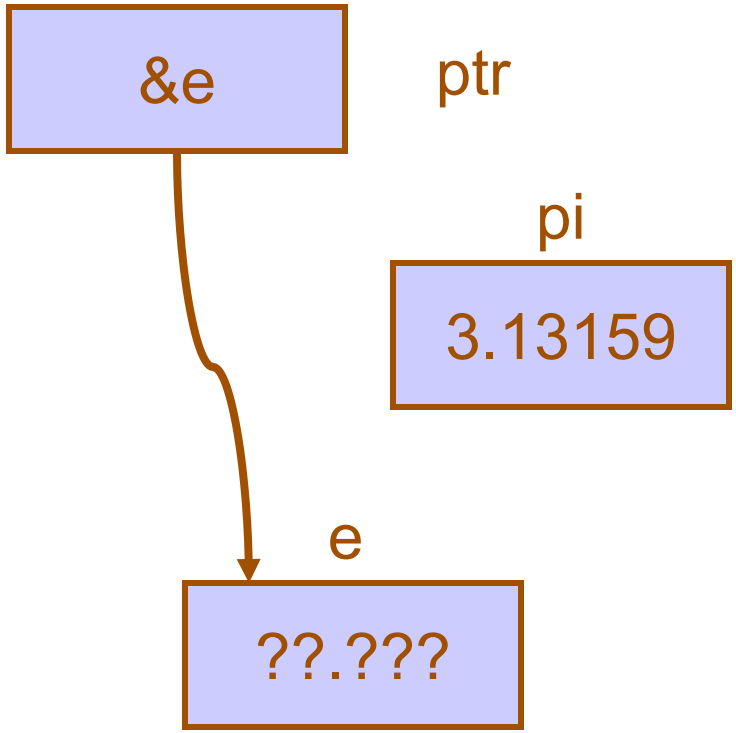
# Pointer Syntax -- Example

```
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
```



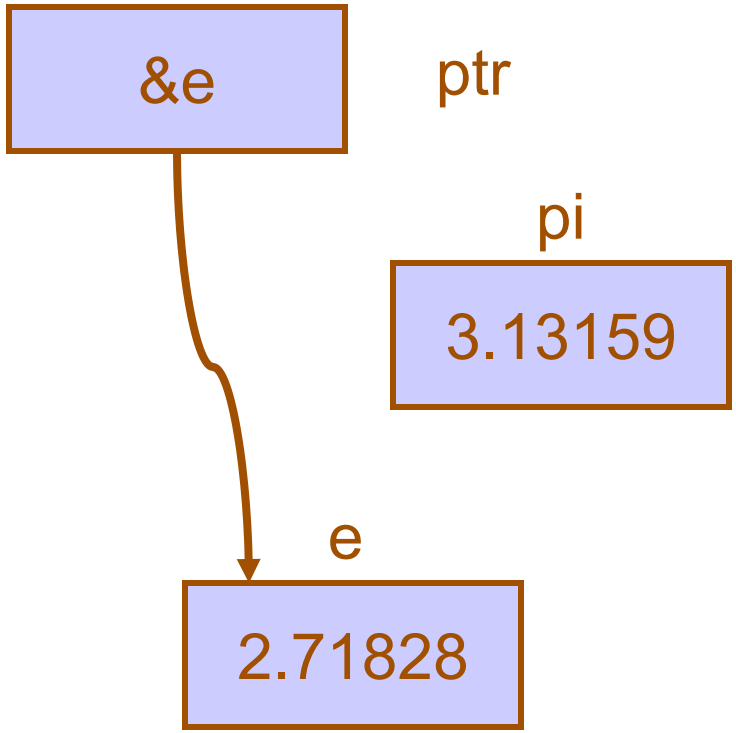
# Pointer Syntax -- Example

```
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
```



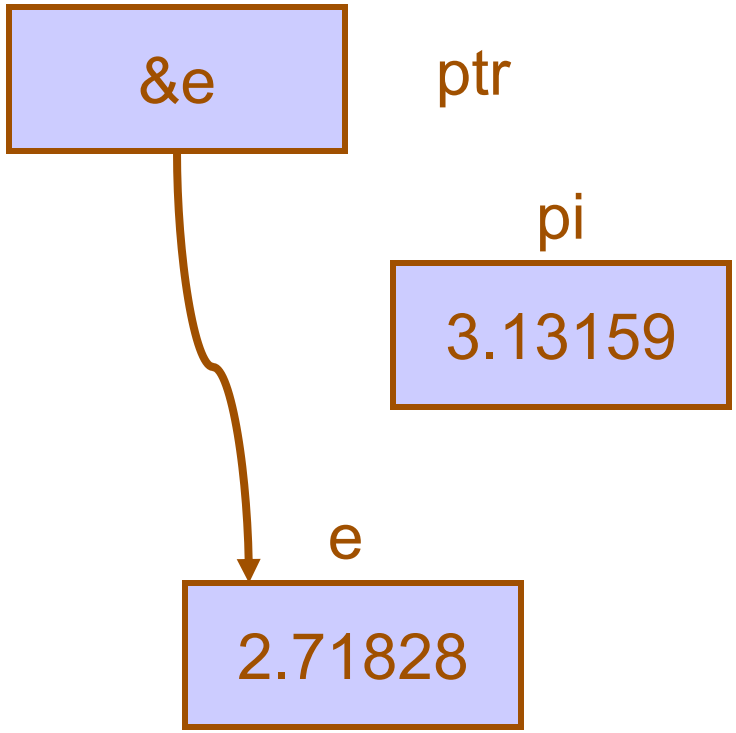
# Pointer Syntax -- Example

```
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
*ptr = 2.71828;
```



# Pointer Syntax -- Example

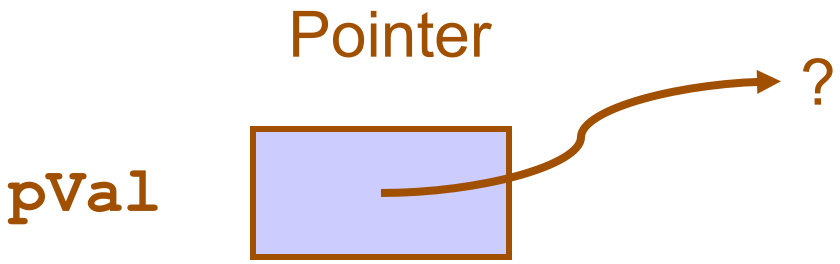
```
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
*ptr = 2.71828;
printf("%p %g %g %g\n",
        ptr,   *ptr,   pi,    e);
```




Output: `ffbff958 2.71828 3.14159 2.71828`

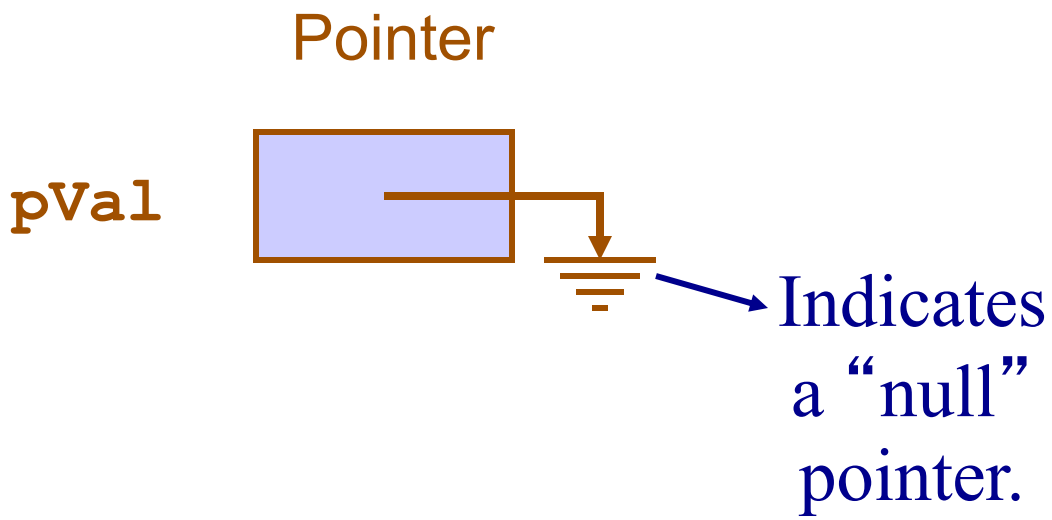
# Pointers – Memory Allocation

```
int *pVal;
```
/* Pointer uninitialized to unallocated integer value. */



# Pointers – Memory Allocation

```
int *pVal;    /* Pointer uninitialized to
                 unallocated integer value. */

pVal = NULL;  /* Initialize pointer to indicate that
                 it is not allocated.  */
```

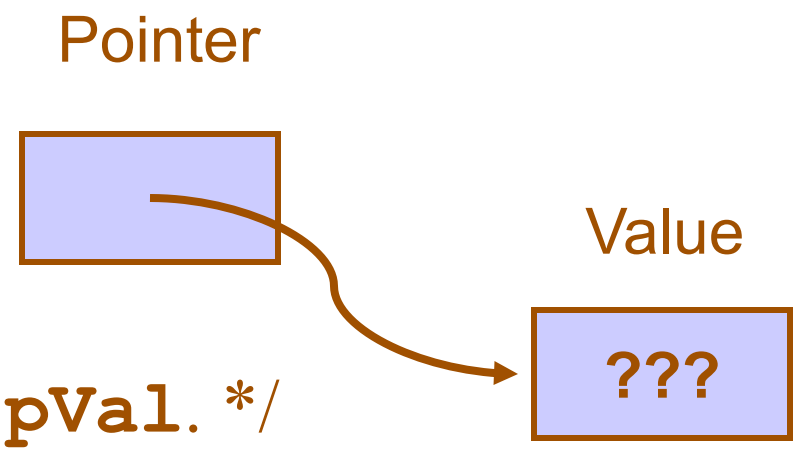


# Pointers – Memory Allocation

```
int *pVal;
```
/* Pointer uninitialized to unallocated integer value. */

```
pVal = 0;
```
/* Initialize pointer to indicate that it is not allocated. */

.
.
.




/* Allocate integer and */
/* assign memory address to **pVal**. */

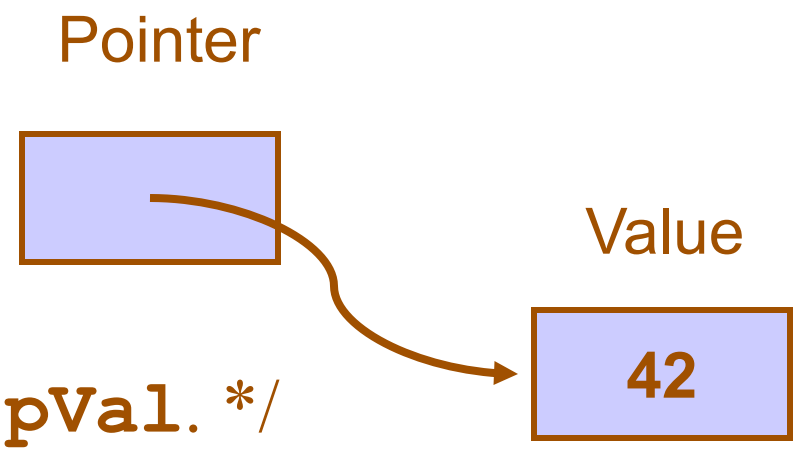
```
pVal = (int *) malloc(sizeof(int));
```

# Pointers – Memory Allocation

```
int *pVal;
```
/* Pointer uninitialized to unallocated integer value. */

```
pVal = 0;
```
/* Initialize pointer to indicate that it is not allocated. */

.
.
.




/* Allocate integer and */
/* assign memory address to **pVal**. */

```
pVal = (int *) malloc(sizeof(int));
*pVal = 42;
```

# Structures

```
struct Gate {
  int  type;       /* Type of gate. */
  struct Gate *left;  /* Left  input.  */
  struct Gate *right;/* Right input.  */
 };
```

# Accessing Data Fields in the Stucture

```
struct Gate gate;

gate.type = 3;
```

but often combined with pointers …

# Pointers and Structures

Pointers often point to structures.

```
struct Gate *p; /* no memory allocated */
struct Gate g;  /* allocates memory */

p = &g;
p->type = 3;/* Set g.type that p points to */
```

# Pointers and Structures

Pointers often point to structures.

```
struct Gate *p;
struct Gate g;

p = &g;
p->type = 3;/* Set g.type that p points to */

/* Same as  (*p).type = 3      */
/* Same as  g.type = 3    */
```

# Dynamic Memory Allocation

- Use **`malloc(num-of-bytes)`**
- Use **`sizeof`** to figure out how many bytes

```
struct Gate *p =
   (struct Gate *) malloc
            (sizeof(struct Gate));

 assert(p != NULL);/* Always check */
```

# Function Arguments

- Pass-By-Value
- Pass-By-Reference

# Function Arguments: Pass-By-Value

- Only values of variables are passed as arguments to a function
- A copy variable is formed in the function
- The value of the argument is lost after returning from the function

# Pass-By-Value -- Example

```
void printing(void){
    int test, n=5;
    test = assignment(n);
    printf("n=%d, test=%d",n,test);
}

int assignment(int n){  /* pass n by value */
    n++;
    return n;
}
```

Output: ?

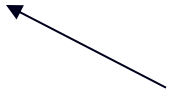
# Function Arguments: Pass-By-Reference

- Pointers to variables are passed as arguments to a function

- The value of the argument is NOT lost after returning from the function, since its memory location is known and stored in the pointer

# Pass-by-Reference -- Example

```
void set_pi(double *p) {
   *p = 3.14159;
}
.
.
double d = 2.718281;
set_pi(&d);  /* Pass d by reference */
printf("d = %g\n", d);
```



pointer

Output: ?

# Structures and Pass-by-Reference Parameters

Very common idiom:

```
struct Vector vec;/* Note: not pointer */


/* Pass by reference */
vectorAdd (&vec, 3.14159);
```

# Arrays Always Passed by Reference

```
void foo(double d[]){/* Same as foo(double *d) */
   d[0] = 3.14159;
}
.
.
double data[4];
data[0] = 42.0;
foo(data);          /* Note: No ampersand. */
printf("%g", data[0]);
```

# Next Lecture

- Read Chapter 5 on ADTs
- Big-OH and Algorithms
- See posted reading and worksheets