CS 261 – Data Structures

C Pointers Review

Oregon State
UNIVERSITY

Object Oriented vs. Procedural

In OOP (e.g. Java), we define classes with methods and call methods 'on' class instances

```
student s = new Student();
s.print();
```

In C, we define functions and in order to ***modify the structure*** with that function, we must pass the structure into the function

```
void printStudent(struct Student myStudent)
{… /* Code to print a single student  struct*/
}
…
struct Student s;
… /*fill s */
printStudent(s)
```

There's one problem however…

C is pass-by-value !!!

C is Pass By Value

Pass-by-value: a copy of the argument is passed in to a parameter

C is Pass By Value

Pass-by-value: a copy of the argument is passed in to a parameter

```
void foo (int a)
{
        a = a + 2;
}
…
void main (int argc, char **argv)  {

{
        int b = 6;
        foo(b)
        printf("b = %d\n", b);
}
```

parameter

argument

Question: What is the output?
Answer: >> b = 6
What if we want to change b?

Oregon State UNIVERSITY

Simulation of Pass-By-Reference

C is Pass-by-value: a copy of the arguments are passed in to a parameter

Changes made inside are not reflected outside

What if we want to change a parameter?

We simulate what is often called “Pass-By-Reference”

To do so, we need to learn about ***Pointers***

Pointers

A pointer is simply a value that can refer to another location in memory

In other words, its value is an address in memory!

Declaring a Pointer (*)

```
int *pVal;
```

Initializing a Pointer

```
pVal = 0;    /* 0 means uninitialized */
```

Get address of (or pointer to) a stored value (&)

```
int a = 5;
pVal = &a;
```

Dereferencing a Pointer (*)

```
*pVal = 4;          /* Assignment*/
int b = *pVal;      /* Access */
```

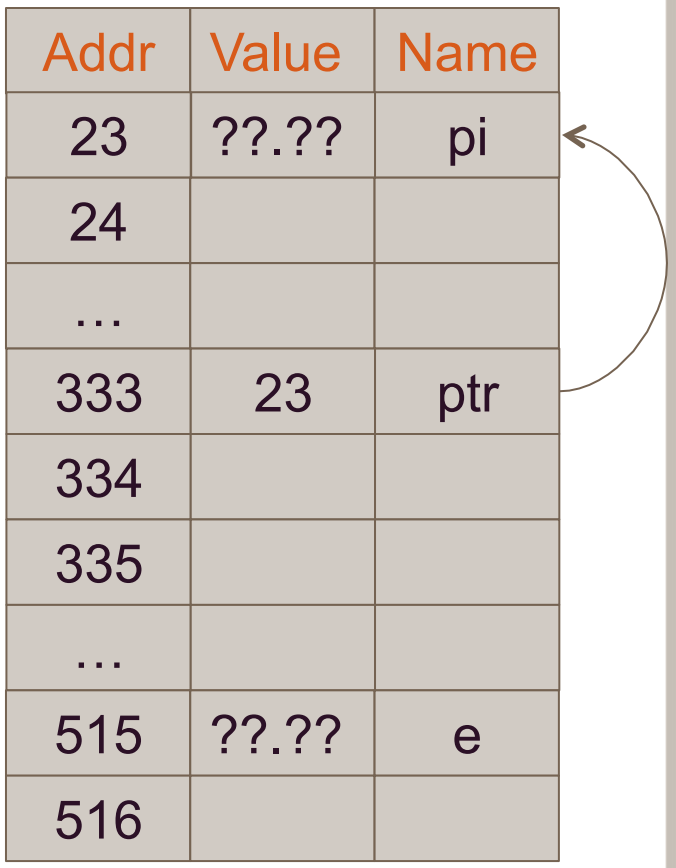
Pointer Example

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

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

Addr	Value	Name
23	??.??	pi
24		
…		
333	?	ptr
334		
335		
…		
515	??.??	e
516		

Pointer Example

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

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

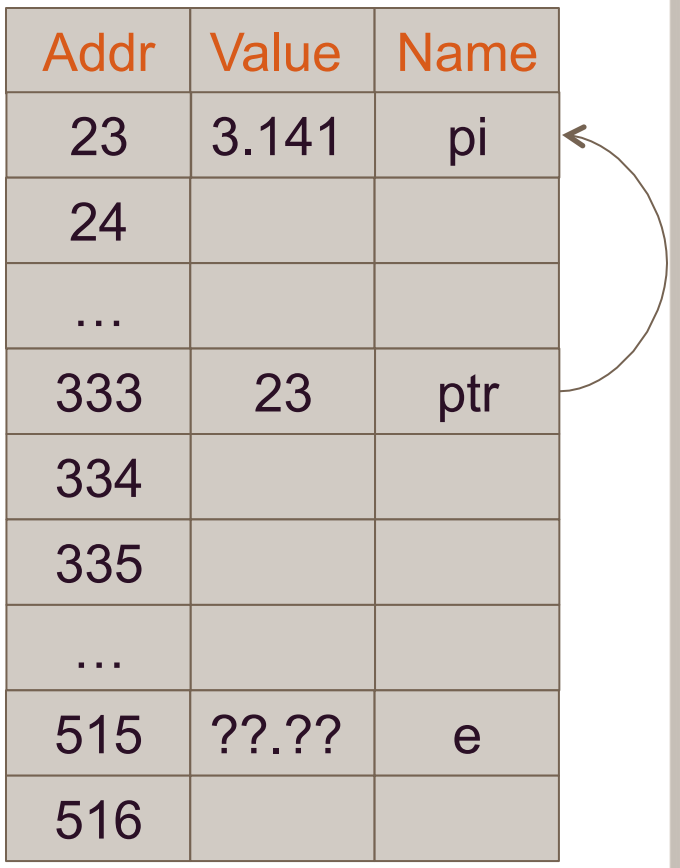


Addr	Value	Name
23	??.??	pi
24		
…		
333	23	ptr
334		
335		
…		
515	??.??	e
516		

Oregon State UNIVERSITY

Pointer Example

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

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



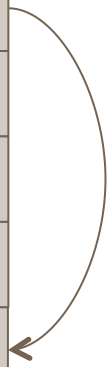
Addr	Value	Name
23	3.141	pi
24		
…		
333	23	ptr
334		
335		
…		
515	??.??	e
516		

Oregon State UNIVERSITY

Pointer Example

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

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

Addr	Value	Name
23	3.141	pi
24		
…		
333	515	ptr
334		
335		
…		
515	??.??	e
516		



Oregon State UNIVERSITY

Pointer Example

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

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

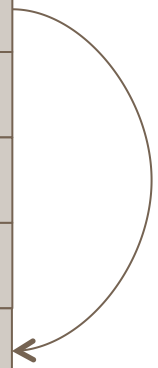
Addr	Value	Name
23	3.141	pi
24		
…		
333	515	ptr
334		
335		
…		
515	2.718	e
516		

Oregon State UNIVERSITY

Pointer Example

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

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

Addr	Value	Name
23	3.141	pi
24		
…		
333	515	ptr
334		
335		
…		
515	2.718	e
516		



```
>> Values: 0x203   2.718   3.141   2.718
```

Oregon State UNIVERSITY

Pass-By-Reference Simulation

Main Idea: If I can pass an address (ie. a pointer), I can't modify it, however, I can modify what it points to (or references)!

```
void foo (int *a)
{
        *a = *a + 2;
}
…
void main (int argc, char **argv)

{
        int b = 6;
        foo(&b)
        printf("b = %d\n", b);
}
```

Addr	Value	Name
23	6	b
24		
…		
333	23	a

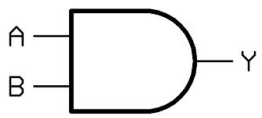
Question: What is the output?
Answer: >> b = 8

Oregon State UNIVERSITY

Pointers and Structures

Pointers often point to structures. Introduces some new syntax:

```
void setGateType(struct Gate *g, int gateVal)
{
        (*g).type = gateVal;
}
```

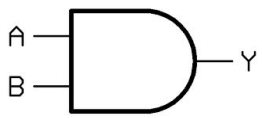



```
struct Gate {
        int          type;
        struct Gate *left;
        struct Gate *rght;
};
```

Pointers and Structures

Pointers often point to structures. Introduces some new syntax:

```
void setGateType(struct Gate *g, int gateVal)
{
        g->type = gateVal   /* equiv to (*g).type */
}
```




```
struct Gate {
        int          type;
        struct Gate *left;
        struct Gate *rght;
};
```

Oregon State UNIVERSITY

Structures and Pass-by-Reference Parameters

Very common idiom:

```
struct Vector vec;      /* Note: value, not pointer. */
vectorInit(&vec);       /* Pass by reference. */
vectorAdd (&vec, 3.14159);

/*or*/
struct Vector *vec2;
vec2 = createVector();  /* returns pointer to
                           struct Vector */
vectorAdd(vec2, 3.1459);
```

Static Memory Allocation

If I know exactly what I need at compile time, I can use static allocation.

e.g. If I need a single struct gate or 5 struct gates

```
struct Gate p;

or

struct Gate p[5];

or

struct Gate p1;

struct Gate p2;

…
```

Dynamic Memory Allocation

But, what if I don't know at compile time?

e.g. I need *N* gates?...where N will be provided as a command line argument or where the user would request one at a time ?

```
/* N gates at once */
struct Gate *p = malloc(N * sizeof(struct Gate));
```

Dynamic Memory Allocation

No **new** operator

Use **malloc(num-of-bytes)** instead

malloc always returns a pointer

Use **sizeof** to figure out how big (how many bytes) something is

```
struct Gate *p = malloc(sizeof(struct Gate));
assert(p != 0);    /* Always a good idea. */
p->type = 3;                    /* safe!*/
…
free(p);
```

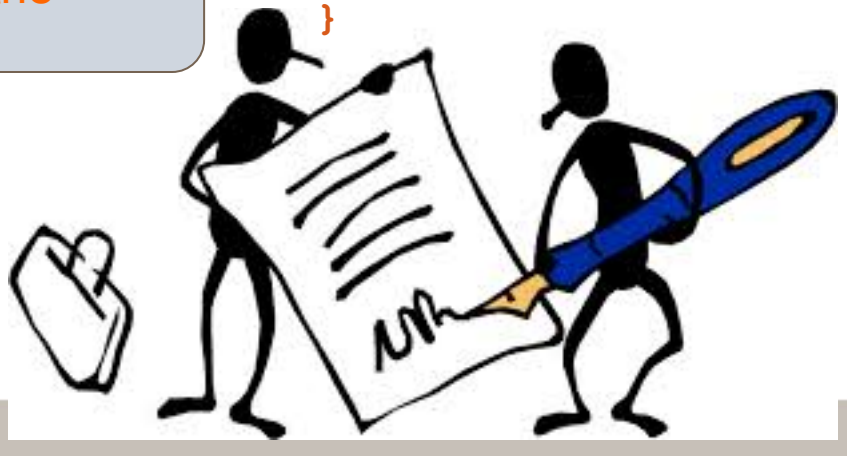
Preconditions, Postconditions & Assert

preconditions are input conditions for the function

postconditions are output conditions for a function

Together, they form a contract between the caller and callee !

```
/*
  pre: size < SIZELIMIT
  pre: name != null;
  post:  result >= MINRESULT
*/
int magic (int size, char *name)
{
        assert(size < SIZELIMIT);
        assert(name != null)
        … DO STUFF …
        assert(result >= MINRESULT);
        return result;
}
```



Oregon State UNIVERSITY

Preconditions, Postconditions & Assert

Practice 1: List preconditions in the header for the function

Practice 2: Calling function should make sure preconditions are met when called

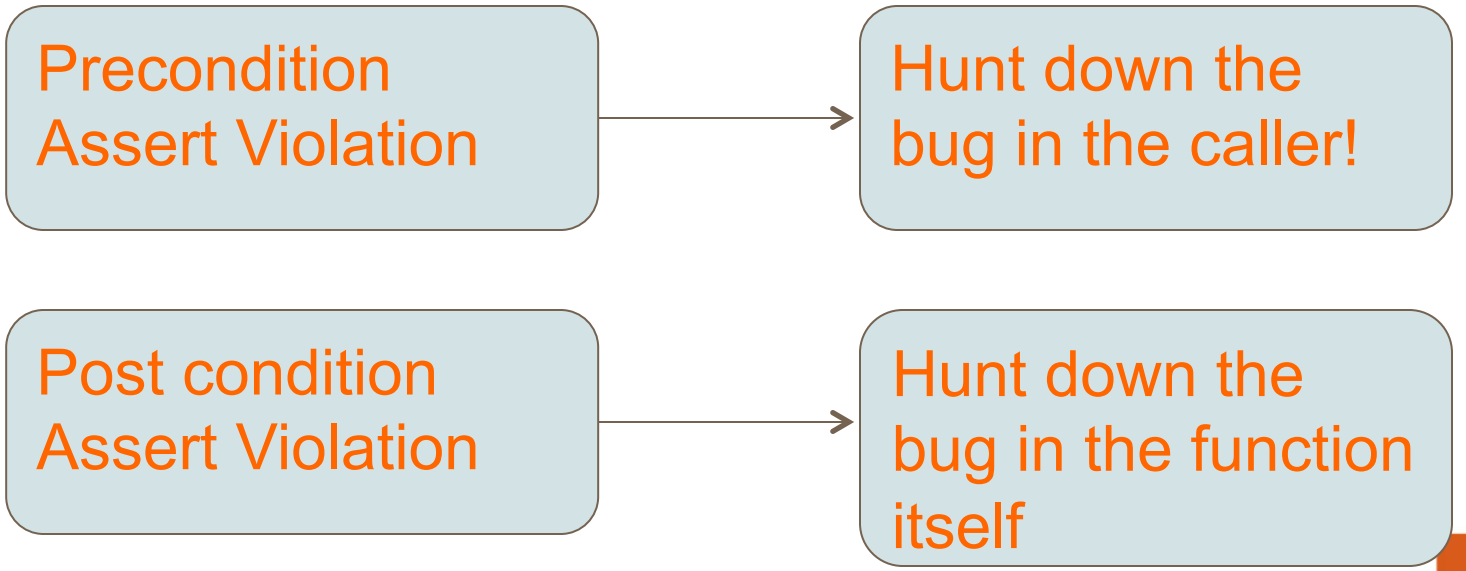
```
/*
  pre: size < SIZELIMIT
  pre: name != null;
  post:  result >= MINRESULT
*/
int magic (int size, char *name)
{
        assert(size < SIZELIMIT);
        assert(name != null)
        … DO STUFF …
        assert(result >= MINRESULT);
        return result;
}
```

```
void foo( char *name ){
        assert( name!= null);
        ...
        magic(  aSize, name);
        ...
}
```

Oregon State UNIVERSITY

Bugs and Errors

1. Program Error: a bug, and should never occur
2. Run-time Error: can validly occur at any time during execution (e.g. user input is illegal) and should be 'handled'



Arrays

Arrays in C are (more or less) pointers

```
        void foo(double d[]) {  /* Same as foo(double *d). */
          d[0] = 3.14159;
        }
        …
        double data[4];  /*static*/
or      double * data = malloc(4*sizeof(double)); /*dyn*/
        data[0] = 42.0;
        foo(data);          /* Note: NO ampersand. */
        printf("What is data[0]? %g", data[0]);
```

Arrays

```
int a[10]
int *pa;
```

a is a pointer to the first element of the array

a[i] refers to the i-th element of the array

pa=&a[0] makes pa point to element 0 of the array, in other words, pa = a

a[2] is the same as *(pa+2) [why? Hint: Contiguous Mem]

one difference: a pointer is a variable, but an array name is not

```
pa = a;        //legal
pa++;          //legal
a = pa;        //not legal
a++;           //not legal
```

Side Note: Booleans

C versions (pre C99) did not have a boolean data type

Can use ordinary integer: test is zero (false) or not zero (true)

Can also use pointers: test is null/zero (false) or not null (true)

```
int i;
if (i != 0) ...
if (i) ...   /* Same thing. */

double *p;
if (p != 0) ...
if (p) ...   /* Same thing. */
```

In C99, we can use bool, but must include header <stdbool.h>

Side Note: Uninitialized Pointers

What if I don't init a pointer, and then access it?

```
struct Gate *p;
/* If external to function, initialized to 0 */
/* If automatic (e.g. local vars), undefined */
p->type = 4;  /* Either way…segmentation fault error
                 Always init for safety …to
                 either value or 0 */
```