CS 261 – Data Structures

C Pointers Review

C is Pass By Value

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

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

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

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*

Oregon State
UNIVERSITY

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 */
```

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	??.??	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		

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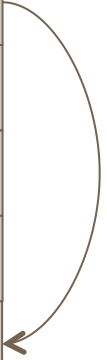
Addr	Value	Name
23	3.141	pi
24		
...		
333	23	ptr
334		
335		
...		
515	??.??	e
516		

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		

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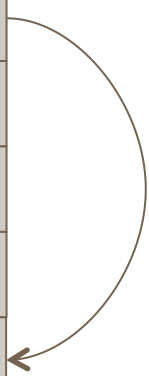
Addr	Value	Name
23	3.141	pi
24		
…		
333	515	ptr
334		
335		
…		
515	2.718	e
516		

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);
    }
}
```
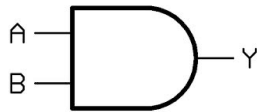
Question: What is the output?
Answer: >> b = 8

Pointers and Structures

Pointers often point to structures. Introduces some new syntax:

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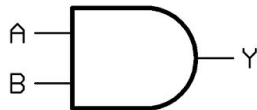


```c
struct Gate {
    int         type;
    struct Gate *left;
    struct Gate *rght;
};
```

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;
};
```

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);
```

Dynamic 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;
```

14

```
...
```

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));



/* One gate at a time */
struct Gate *p = malloc(sizeof(struct Gate));
```

Dynamic Memory Allocation

```
struct Gate *p;

p->type = 4;   /* segmentation fault error */



/* static allocation */

struct Gate p;

initGate(&p);  /*Fills the gate fields */
```

Oregon State
UNIVERSITY

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);
```

Oregon State
UNIVERSITY

Check Conditions: assert

We will use **assert** to check all sorts of conditions

Halts program if condition not found

```
#include <assert.h>


/* Assert checks if specified condition is true. */
assert(whatever-condition);
```

Oregon State
UNIVERSITY

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*/
double data* = malloc(4*sizeof(double)); /*dyn*/
```

or

```
data[0] = 42.0;

foo(data);           /* Note: NO ampersand. */

printf("What is data[0]? %g", data[0]);
```

Oregon State
UNIVERSITY

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)

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: Unitialized Pointers

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

```
struct Gate *p;

p->type = 4;   /* segmentation fault error */
```

Oregon State
UNIVERSITY