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) {
    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 = 6
What if we want to change b?
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 (*)

\[
\text{int } *\text{pVal};
\]

Initializing a Pointer

\[
\text{pVal } = 0; \quad /* 0 \text{ means uninitializ}ed */
\]

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

\[
\text{int } a = 5;
\text{pVal } = &a;
\]

Dereferencing a Pointer (*)

\[
*p\text{Val } = 4; \quad /* \text{Assignment}*/
\text{int } b = *\text{pVal}; \quad /* \text{Access }*/
\]
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);
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);
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);
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);
**Pointer Example**

```c
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);
```

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>3.141</td>
<td>pi</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>333</td>
<td>515</td>
<td>ptr</td>
</tr>
<tr>
<td>334</td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>515</td>
<td>2.718</td>
<td>e</td>
</tr>
<tr>
<td>516</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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);

>> Values: 0x203  2.718  3.141  2.718
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)!

```c
void foo (int *a)
{
    *a = *a + 2;
}
...
void main (int argc, char **argv)
{
    int b = 6;
    foo(&b)
    printf("b = %d\n", b);
}
```

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>6</td>
<td>b</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>333</td>
<td>23</td>
<td>a</td>
</tr>
</tbody>
</table>

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;
}
```

```
struct Gate {
    int type;
    struct Gate *left;
    struct Gate *right;
};
```
Pointers and Structures

Pointers often point to structures. Introduces some new syntax:

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

```c
struct Gate {
    int type;
    struct Gate *left;
    struct Gate *right;
};
```
Structures and Pass-by-Reference Parameters

Very common idiom:

```c
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($N \times \text{sizeof(struct Gate)}$);
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

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

We will use `assert` to check all sorts of conditions. Halts program if condition not found.

```c
#include <assert.h>

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

Arrays in C are (more or less) pointers

```c
void foo(double d[]) { /* Same as foo(double *d). */
    d[0] = 3.14159;
}
...

double data[4]; /*static*/
dooble * data = malloc(4*sizeof(double)); /*dyn*/
data[0] = 42.0;
foo(data); /* Note: NO ampersand. */
printf("What is data[0]? \%g", data[0]);
```
Arrays

```c
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.

```c
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)

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

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

```c
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 */
```