Test 1

Friday, February 25th

Wednesday: Review

Email me questions/topics to cover.
Pointer practice

```c
int j = 3, k=5, *p = &j, *q = &j,*r;
double x;
p == &j
**&p
r =&x
7 * *p / *q +7
*(r = &k) *= *p
```

1
3
illegal
14
15
Null Pointers

- Have a value of 0

```c
int *p;
p = 0;
p = NULL; //pre-defined macro.
```

- Means the pointer is inactive
void pointers

- Generic pointer type
- can be used as gateways
- don’t have to cast

```c
void *foo() { . . . }
int *p = foo();
char *q = foo();
```
Misc. Pointer stuff

- Can’t point to register variables
  - register int v; &v;
- Can’t point to constants
  - &3
- Can’t point to ordinary expressions
  - &(k+99);
Pointer to Pointer

Called Multiple indirection

q → \text{address} → p → \text{address} → x → \text{value}

**q → *p → x
```c
#include <stdio.h>
void main(void) {
    int x, *p, **q;
    x = 10;
    p = &x;
    q = &p;

    printf("%d", **q);
}
```
Arrays of pointers

- argv is an example

- int *x[10]; // creates array of 10 pointers

- rules the same as before
  - x[2] = &var
  - *x[2] returns value from that pointer
Multidimensional pointers

ch[][];

**ch == ch[0][0];
**(ch + 1) == ch[1][0];
(*ch + 1) == ch[0][1];
(*((ch+1)+1) == ch[1][1];

Really messy, best leave it alone
Using `argv` as pointer

```c
main(int argc, char *argv[]) ==
main(int argc, char **argv)
```

*`argv` == first string in the array.
**`argv` == first character;

`++argv` := moves to next string.
```c
#include <stdio.h>
int main(int argc, char **argv) {
  char **p = argv+1;
  while (*p) puts(*p++);
  return 0;
}

int main(int argc, char **argv) {
  ++argv;
  while (*argv) puts(*argv++);
  return 0;
}
```
int main(int argc, char **argv) {
    char *p;
    int j;

    for (j = 1; j < argc; ++j){
        p = argv[j];
        puts(p);
    }
    return 0;
}
Call by reference

- pass pointers as arguments
  ```c
  void foo(int *p) {
  }
  
  int x = 3; *q = &x;
  foo(&x);
  foo(q);
  
  foo has access to x
  q != p. Separate pointers to same location
  ```
Dynamic Memory

- allocate memory on the fly
- array size doesn’t have to hard coded
- Useful in data structures
- No constructors in C
malloc & calloc

- in stdlib.h

- will allocate memory chunk

- returns pointer to start of chunk
void *calloc(size_t num, size_t size)

- initializes allocated memory to 0
- num is number of desired elements
- size is the number of bytes per element
  - char = 1, int = 2-4, etc
void *malloc(size_t size)

does NOT initialize memory.

Is faster.

size is the total bytes to be allocated

malloc( n * bytes)
sizes

- Vary from machine to machine
- Not fair to make you memorize them
- C makes it easy
malloc

Do you wanna allocate a 10-int array?

```c
int *p;
p = (int *) malloc(400); // int is 4 bytes
```
sizeof operator

- returns size of input type

sizeof(char)
sizeof(int)

int *p;
sizeof(p);
Back to malloc

```c
void *malloc(size_t size)

size = number of elements * size of elements
  = n * sizeof(type)

array of 10 ints:
  int *p = malloc(10 * sizeof(int));

array of 15 floats:
  float *p = malloc(15 * sizeof(float));
```
Malloc/Calloc

- Returns 0 if error
- Not enough memory error is common

```c
int *p;
if (!(p = (int *) calloc(10, sizeof(int))))
    printf(“out of memory”);
```
Drip Drip Drip Drip

```c
int *p;
p = (int *) malloc(10 * sizeof(p));
//do some stuff here

p = (int *) malloc(20 * sizeof(p));

????
```
Memory Leaks

- Operating system usually deals after program execution
- Problem for big programs
- Must find a way to get used memory back
void free(void *ptr)

- in stdlib.h
- frees up chunk pointed to by ptr
- destroys all info in chunk
free(void *ptr)

IMPORTANT!!!!!

Only give pointers that were allocated with either malloc or calloc

Do this or else. Seriously. ptr must be exact pointer place given in allocate function
free

```c
int *p;
p = (int *)malloc(10 * sizeof(p));
//do some stuff
free(p); // p must still point to old address
p = (int *) malloc(10 * sizeof(p));
```