LECTURE 18: POINTERS / ARRAYS (K&R §§ 5.4, 5.5, 5.11)

malloc() and free()

simple description:

To get a pointer p to a block of memory that is n characters in length, we can call:

```
  p = malloc(n);
```

When we're done with the memory, we need to return it by calling:

```
  free(p);
```

The memory address returned by malloc() is located in the heap.

malloc

A void pointer is a general purpose pointer that does not have a data type associated with it.

malloc returns a void pointer (void *) that points to a memory block of n bytes.

To request a pointer to n of a specific type:
-request a memory block in sizeof the type
-cast the pointer returned by malloc

```
  int *p;
  p = (int *) malloc(n * sizeof(int));
```

If it is unable to provide the requested memory, malloc returns a NULL pointer value.

If you dereference a NULL pointer to access memory: system crash!

Always check to be sure that the pointer returned by malloc is NOT equal to NULL.

If a pointer is NULL, code must take appropriate recovery action to handle the lack of memory.

free

The call to free does not clear the program's pointer to the memory block, so it is now a stale pointer.

If a program uses a pointer after free by accessing or setting memory via pointer, it could overwrite data owned by another program: system crash!

If program calls free again with the same pointer, it releases memory possibly owned by a different program now: system crash!

You should set a pointer to NULL after calling free for the above reasons.

memory leaks

If you set the pointer to a memory block to NULL before calling free, you have caused the system to lose the memory forever.

If this happens enough times: system crash!

MUST NOT clear or overwrite a pointer to a memory block before calling free!

memory model for malloc() and free()

Before call to malloc():

allocbuf:	free

After 3 calls to malloc():

allocbuf:	in use	in use	in use	free

After call to free()on second allocated block of memory:

allocbuf:	in use	free	in use	free

After another call to malloc():

allocbuf:	in use	free	in use	in use	free

...

Fragmentation after several calls to malloc() and free():

allocbuf:	in use	free	in use	free	in use	free	in use

At this point, malloc cannot provide a large contiguous block of memory - even though there is theoretically sufficient free memory. The memory is fragmented.

This is a difficult problem to solve.

It's not possible to "defragment" malloc memory as you would do for a disk.

On a disk, the pointers to memory are in the disk file allocation table and can be changed.

With malloc, programs are holding pointers to memory they own, so they can't be changed.

POINTERS TO FUNCTIONS

We've seen pointers to variables, but C also allows pointers to functions.

Say we've defined two functions: inc and dec.

```
void inc(int *num) {
  (*num)++;
}
```

```
void dec(int *num) {
  (*num)--;
}
```

Now we'll define a function pointer. We want to point it to the functions above, so we'll define a pointer for a function that has no return value (void) and receives an int pointer:

```
void (*fp)(int *);
```

We can define a function that accepts our function pointer as an argument:

```
void change(void (*func)(int *), int *num) {
  (*func)(num);
}
```

Now we can call change with inc or dec as an argument and change will execute the corresponding function.

e.g., we could do something like:

```
int count = 0;
fp = NULL;
while ( (c = getchar()) != EOF) {
  switch (c) {
    case ('i'):
      printf("Switched to increment mode.\n");
      fp = &inc;
      break;
    case ('d'):
      printf("Switched to decrement mode.\n");
      fp = &dec;
      break;
    case ('a'):
      printf("Applying current mode to count.\n");
      if (fp != NULL) {
        change(fp, &count);
        printf("count = %d\n", count);
      } else {
        printf("Must enter inc or dec mode to apply.\n");
      }
      break;
  }
}
```

Function declaration for qsort (K&R § 5.11):

```
  void qsort(void *lineptr[], int left, int right,
             int (*comp)(void *, void *));
```

There are four arguments:

lineptr	an array of void pointers
left	an int
right	an int
comp	a pointer to a function that returns an int and takes two arguments that are pointers

This last argument allows us to pass a custom comparison function to qsort, which it will then use to sort.

If we want to use qsort to sort strings and we have a string comparison function called strcmp, we can use the following call:

```
qsort((void**) lineptr,
      0,
      nlines-1,
      (int (*)(void*,void*))(strcmp));
```

Here we cast to a function pointer to strcmp. The & operator is not required for strcmp as it names the address of a function (similar to passing array names).

Within qsort(), function is called via pointer:

```
  if((*comp)(v[i], v[left]) < 0) ...
```

initialize a pointer to a function

```
/* function pointer *fooptr = cast of foo to funct ptr */
int(*fooptr)(int *, int *) = (int (*)(int *, int *))foo;
```

call the function foo via the pointer to it

```
(*fooptr)(to, from);
```

LECTURE 18: STRUCTS (K&R § 6.1)

STRUCT BASICS

struct	A collection of variables, possibly of different types, grouped under a single name for common reference as a unit.

Example

To represent a point in a two-dimensional graph, we can declare a point structure:

```
  struct point {  /* with optional structure tag (name) */
    int x;        /* member x */
    int y;        /* member y */
  };
```

A struct declaration defines a type.

The closing brace may be followed by a list of variables, just as for any basic type:

```
  struct point
    int x;
    int y;
  } pt1, pt2, pt3;
```

syntactically similar to:

```
  int i1, i2, i3;
```

Both declare variables of the named type and set space aside for them.

Now we can define variables using our struct:

```
  struct point p4, p5, p6;
```

And we can initialize the members when we define it:

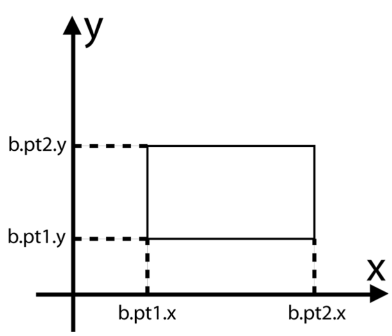
```
  struct point p7 = {5, 10};
```

We can assign values to members after definition using the structure member operator, or dot operator:

```
  p7.x = 10;
```

We can also nest structs. For example, we can use our point structure to define a rectangle structure:

```
  struct rect {
    struct point pt1;  /* lower left  */
    struct point pt2;  /* upper right */
  };
```



We can then define a function that returns the area of the rectangle using the struct members:

```
  int rectarea(struct rect b) {
    return (b.pt2.x - b.pt1.x) * (b.pt2.y - b.pt1.y);
  }
```