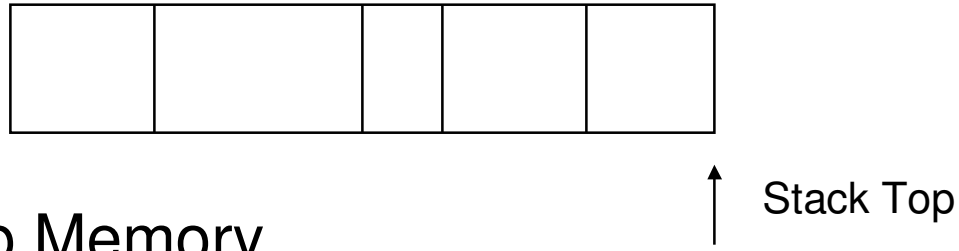
Announcements

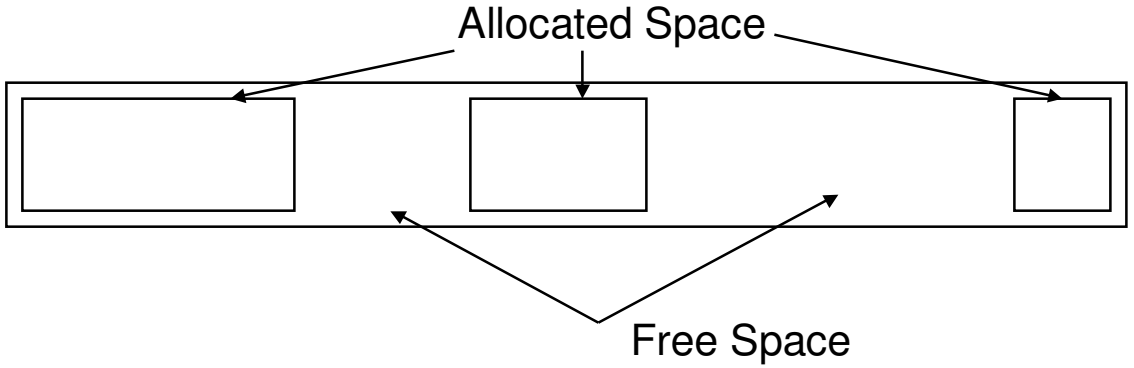
- Program #2
  - Due Date Extension: Tuesday, March 15, 2005
- Reading
  - Chapter 11 (Today)
  - Chapter 14 (Thursday)

Stack and Heap Memory

- Stack Memory
  - Allocated and de-allocated in a specific order
    - last allocated is the first to be freed
  - Useful for representing program local variables



- Heap Memory
  - Can allocate and de-allocate in any order



Dynamic Memory

- Why Dynamic Memory
  - Eliminate Compile Time Limits
    - hard to get sizes of things exactly right
    - over estimation leads to wasted space
    - under estimation leads to runtime errors or falling off into non-allocated space
- User Managed Heap vs. Garbage Collection
  - Garbage Collection
    - Don't need to worry about de-allocation
    - Takes times to run collection algorithms
  - User Managed
    - Control precisely when to de-allocate an object
    - Can create errors if de-allocate a object still in use
    - Can run out of space on the heap if you do not de-allocate

Heap Memory Management

- Allocating memory:
  - **void *malloc(size_t requestedSize);**
    - allocate requestedSize bytes of memory
    - no guaranteed initialization of that memory space
  - **void *calloc(size_t requestCount, size_t objectSize);**
    - allocate requestCount objects of size objectSize bytes each
    - set the requested memory to zero
- Type Information
  - malloc/calloc return void *
  - need to convert to desired type using cast operator

Example of malloc

```
#include <malloc.h>
#include <stdio.h>

int main(void){

T *p;
p = (T*) malloc(sizeof(T));
if (!p) {
   exit(-1);   /* often nothing can do on failure */
}
/* use of p */

}
```

Example of calloc

```
#include <malloc.h>
#include <stdio.h>

void dosomething(int k) {
    int i, *myInts;
    myInts = (int *) calloc(k, sizeof(int));
    if (!myInts) {
        printf("no memory\n");
        exit(-1);
    }
    for (i=0; i < k; i++) {
         myInts[i] = anotherFunc(i, k);
    }
}
```

Memory De-allocation

- free(void *p);
  - returns space pointed to by p
  - does **not** change p
    - good idea p=NULL; right after de-allocation.
- Using Free
  - always free memory when done with it
  - never free things that are not allocated by malloc/calloc:
    - int a = 3;
    - int *p = &a;
    - free(p); /* error, p points to something not allocated */
  - Never dereference a pointer after free is called:
    - free(p);
    - p = NULL;
    - if (*p) { …} /*runtime error, pointer is no longer valid */

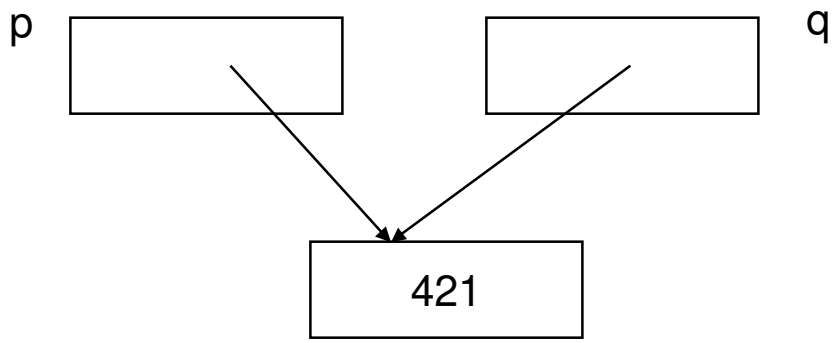
What if you need more memory?

- Could do multiple steps
  - malloc new space
  - copy data from old space to new
  - free old space
- C provides a short cut
  - **void *realloc(void *ptr, size_t new_size);**
  - allocates new space (larger than old)
  - copies old data and frees it
  - returns pointer to new space

Pointer Aliases

- Sometimes two pointers point to the same thing
  - This can be useful
  - This can cause problems:

```
int *p, *q;
p = (int *) malloc(sizeof(int));
*p = 421;
q = p;
free(p);
p = NULL;
if (*q) { … }        /* error: *q points to freed space */
```



Common Dynamic Memory Errors

- Derefencing NULL pointers
  - int *foo = NULL;
  - *foo = 3; /* ERROR – segmentation fault right away*/
- Forgetting to check return value of malloc/calloc
- Using malloc and not initializing values
  - memset
- Going past end of allocated space
  - very easy when using array syntax and loops

```
int i, *myInts;
    myInts = (int *) calloc(k, sizeof(int));
    if (!myInts) {
        printf("no memory\n");
        exit(-1);
    }
    for (i=0; i <= k; i++) {
        myInts[i] = anotherFunc(i, k);
    }
```

Goes past end of array allocated

More Common Memory Errors

- Calling Free on something not from heap

```
int *ptr; j;
j = 3;
ptr = &j;
free(ptr);     /* ERROR: ptr does not point to something in heap */
```

- Calling Free on other than the start of allocation

```
int *ptr;
ptr = (int *) calloc(10, sizeof(int));
free (ptr + 3);
```

- Using memory that has been freed

```
T *curr;
free (curr);
curr = curr->next;     /* ERROR: curr has already been freed */
```

Memory Leaks

- Common problem to forget to free memory
  - Doesn't cause the program to crash (at first)
  - Wastes memory and can cause the program to slow down
- Example:

```
int *myInts;

myInts = (int *) malloc(100);

/* no other pointers used to remember where those 100 bytes
   are and no freeing of the space for those 100 bytes */

myInts = (int *) malloc(200);        /* now leaked 100 bytes */
```

Pointers and Functions

- Functions can return pointer to dynamically allocated memory
  - ---------- good:

```
int *getInt() {
    int *ptr;
    ptr = (int *) calloc(42, sizeof(int));
    return ptr;
}
```

  - ---------- bad:

```
int *getInt() {
    int a=7;
    int *ptr = &a;
    return ptr;
}
```

Other Helpful Memory Routines

- char *strdup(char *string);
  - allocate enough memory to hold string (including it's null character)
  - copy string into the allocated memory
  - #include <string.h>

- void *alloca(size_t size);
  - allocate memory on the **stack**
  - memory is freed when the function returns\
  - #include <alloca.h>

More Dynamic Memory Examples

```
typedef struct {
    int count;
    char *name;
    float price;
} item;
item *inventory;
….
{
char newName[] = “Jan”;
inventory = calloc(sizeof(item), 20);
for (i=0; i < 20; i++) {
     inventory[i].name = strdup(newName);
}
```

Midterm Results

	Q1	Q2	Q3	Q4	Q5	Q6	Total
Avg	11.5	13.7	11.2	11.5	15.1	6.9	69.9
Min	4.0	8.0	0.0	1.0	6.0	0.0	36.0
Max	16.0	15.0	15.0	18.0	21.0	15.0	92.0
Std. Dev	2.9	1.9	3.6	4.1	4.0	3.8	13.0

- Re-grade Requests
  - must be in writing via the submit server
  - deadline is Thursday 3/10/05 1:45 PM
  - must hand-in your exam booklet to me by the deadline
  - you must not alter your exam booklet in any way if you are submitting it for a re-grade request
  - re-grading may result in score going up or down

Common Mt #1 Mistakes

- #1
  - Program Counter
    - advances one location per instruction, branch changes
  - Function Prototype
    - defines name, paramter types, and return types
    - allows compiler to type check functions in separate files
- #2
  - last integer (+2 is the value 4)
- #3
  - Explaining two passes
  - Describing memory field & that a new op-code was needed

Common Mt #1 Mistakes (cont.)

- #4
  - Expressions can be valid even if they don't refer to scalars
- #5
  - finding comma, and not overwriting parts of string that are needed
  - printing lastname firstname not firstname lastname
- #6
  - Make example - go re-read how to create a make file
  - cp copies one file, doesn't combine two
  - Review basic UNIX commands