

# Dynamic Memory Allocation

CSE 142 - Spring Quarter

# Static memory

- Usually you know exactly what you need ahead of time
  - `int x;`
  - `char myString[50];`
  - `double price;`
- This is called static memory allocation

# The stack

- Global variables are declared in memory right next to your program.
- Local variables in functions or main are put on the stack.

# Dynamic Memory

- There's also an area called the "heap"
- You can ask for some memory from the heap.
- When you're done, you give it back.



# Dynamic memory

- getting memory
  - `void * malloc(size_t number_of_bytes);`
- returning memory
  - `void free( void * p);`
- `#include <stdlib.h>` to get these functions

# Getting memory

- malloc needs to know how many bytes you need
- Use sizeof to find the right size, no matter what computer you're on

```
size_t number_of_bytes_in_an_int  
= sizeof(int);
```

# Getting memory

- The `size_t` type is just an integer that can hold the biggest size your computer can allocate.
  - Can be 8 or 16 bits, usually
- The size of each thing is implementation dependent
  - `size_t number_of_bytes_in_an_int = sizeof(int);`

# Getting memory

- malloc returns a void \*
- A void \* just means pointer to anything.

```
int * pint;  
pint = (int *) malloc(sizeof  
    (int));  
// sets aside space for one int
```

# The NULL pointer

- The NULL pointer is always 0
- It points to nothing and always means this pointer is invalid
- You should always check a pointer for NULL before you use it
- malloc returns NULL if it's unsuccessful getting your memory

# The NULL pointer

```
int * place_in_line;  
place_in_line =  
    malloc(sizeof(int) * 500);  
  
if (place_in_line == NULL) {  
    //failed to get memory!  
}
```

# Using malloc

```
int number;  
printf("Enter number of  
grades:");  
scanf("%d", &number);
```

# Using malloc

```
double * pgrades = (double *)  
    malloc(number * sizeof(double));  
  
for(int i=0; i< number; i++) {  
    printf("Enter grade %d:", number+1);  
    scanf("%lf", pgrades+i);  
}
```

# Freeing Memory

When you're done with the memory, give it back

If you don't, or you lose track of the memory, it's called a memory leak!



# Freeing memory

```
int * x = (int *) malloc(sizeof  
    (int));  
int * y = (int *) malloc(sizeof  
    (int));  
x = y;  
// where is x's memory?
```

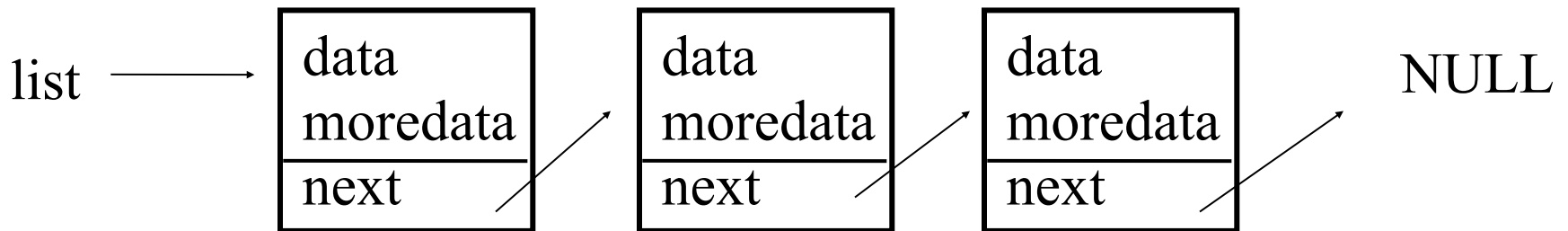


# The Linked List

- The problem : you need an ordered series of objects
- You could allocate them all at once with an array or malloc
- But you won't know how many you'll need til you're through
- What if each object contained a link to the next one?

# The Linked List

```
struct node {  
    int data;  
    double moredata;  
    struct node * next;  
}
```



# The Linked List

```
struct node {  
    double grade;  
    struct node * next;  
}  
  
struct node * start = NULL;  
struct node * current = NULL;
```

# The Linked List

```
while(++counter) {  
    printf("Enter grade or -1 to quit : ");  
    double input;  
    scanf("%lf", &input);  
    if (input < 0)  
        break;  
    struct node * new = (struct node * ) malloc(sizeof  
    (struct node));
```

# The Linked List

```
if (current == NULL) {
    printf("Error allocating memory\n");
    exit;
}
if (start = NULL) {
    start = current = new;
    start->next = NULL;
}
else {
    current->next = new;
    current = new;
}
}
```

# Searching linked list

- Find the highest grade in the linked list
  - start at the beginning, check each node.
  - when you get to NULL, you're done

```
double highest = 0.0;
struct node * current = start;
while (current != NULL) {
    if (node->grade > highest)
        highest = node->grade;
    current = current.next;
}
```

# Sorting linked list

