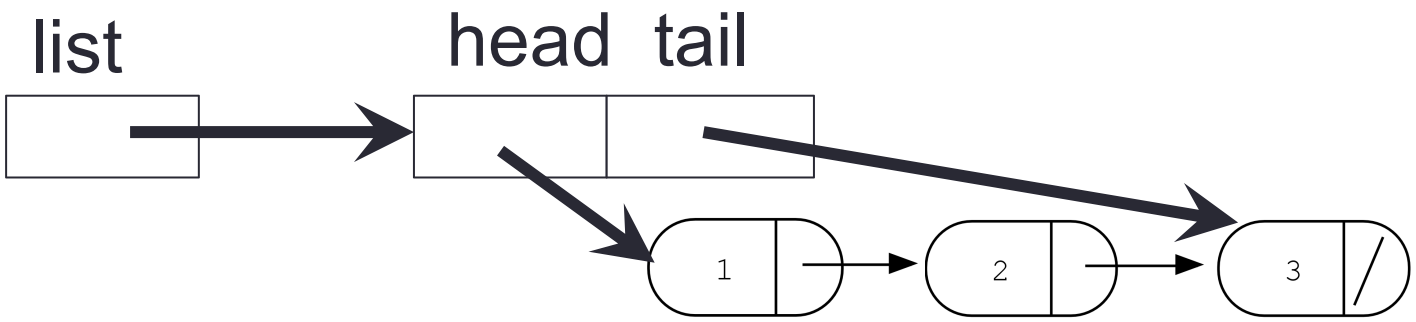
LINKED LISTS (CONTD)
DYNAMIC ARRAYS

Problem Solving with Computers-I

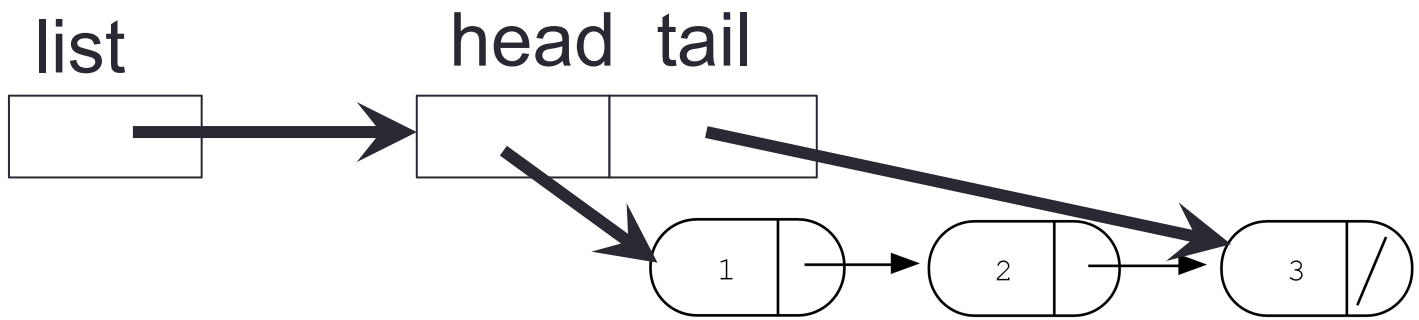
https://ucsb-cs16-wi17.github.io/

Review:
What are the ‘links’ in a linked-list?



Iterating through the list

```
int lengthOfList(LinkedList * list) {
}
```



Dynamic memory pitfall: Memory Leaks

- Memory leaks (tardy free)
  - Heap memory not deallocated before the end of program (more strict definition, potential problem)
  - Heap memory that can no longer be accessed (definitely a leak , must be avoided!)

Does calling foo() result in a memory leak?

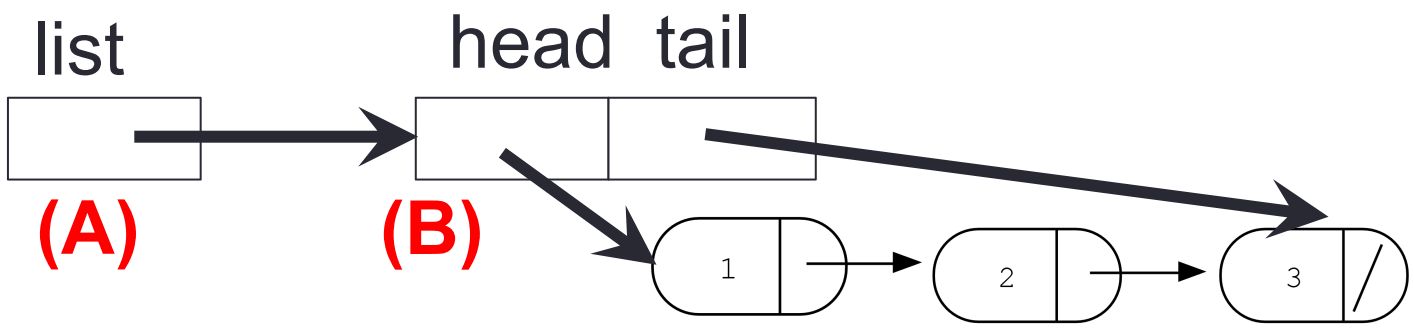
```
void foo(){
     int * p = new int;

}
```

- How to avoid memory leaks?
- How to detect memory leaks?

Deleting the list

```
int freeLinkedList(LinkedList * list){…}
```

Which data objects are deleted by the statement: `delete list;`



(C) All nodes of the linked list

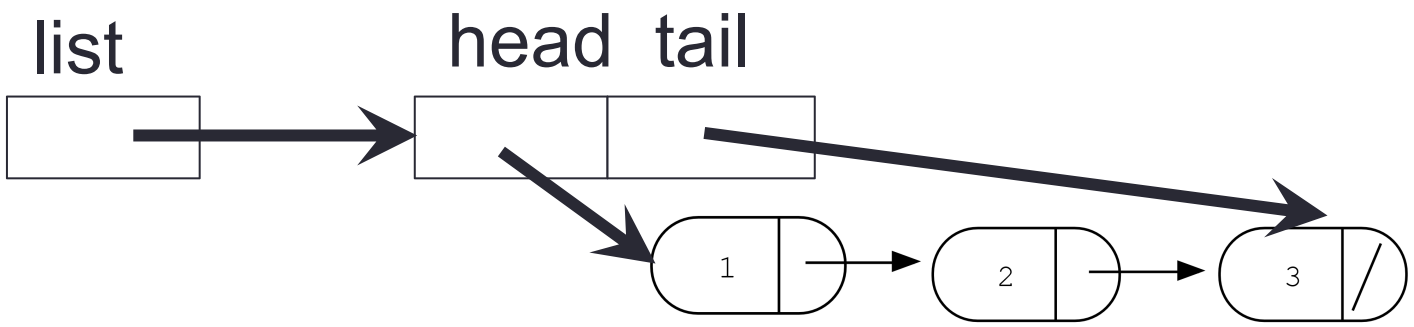
(D) B and C

(E) All of the above

Does this result in a memory leak?

Deleting the list

```
int freeLinkedList(LinkedList * list);
```



Dynamic arrays

```
int arr[5];
```

```
struct UndergradStudents{
    string firstName;
    string lastName;
    string major;
    double gpa[4];
};
```

Arrays and pointers


```
int arr[]= {20, 30, 40, 50, 60}
```

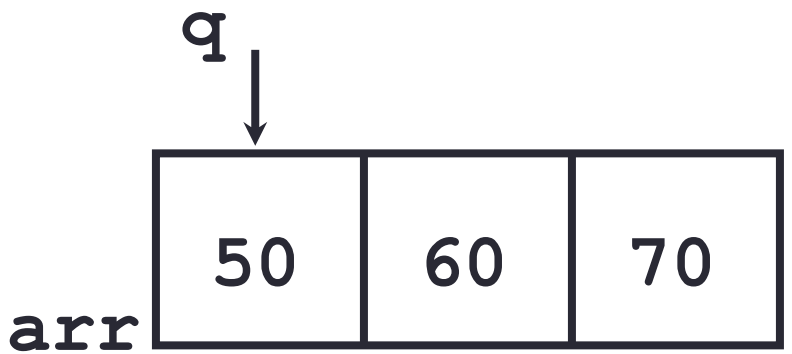
- `arr` is a pointer to the first element, what is the output of `cout<<arr;`
- `arr[0]` is the same as `*arr`
- `arr[2]` is the same as `*(arr+2)`

- Use pointers to pass arrays in functions (See code from lecture 9)
- Use *pointer arithmetic* to access arrays more conveniently

Pointer Arithmetic

```
int arr[]={50, 60, 70};
int *p;
p = arr;
p = p + 1;
*p = *p + 1;

UndergradStudents records[2];
UndergradStudents *pRec;
pRec = records;
pRec = pRec + 1;
```

```
void IncrementPtr(int *p){
    p =  p + 1;
}

int arr[3] = {50, 60, 70};
int *q = arr;
IncrementPtr(q);
```



Which of the following is true after **`IncrementPtr(q)`** is called in the above code:

A. ‘q’ points to the next element in the array with value 60

B. ‘q’ points to the first element in the array with value 50

How should we implement `IncrementPtr()`, so that ‘q’ points to 60 when the following code executes?

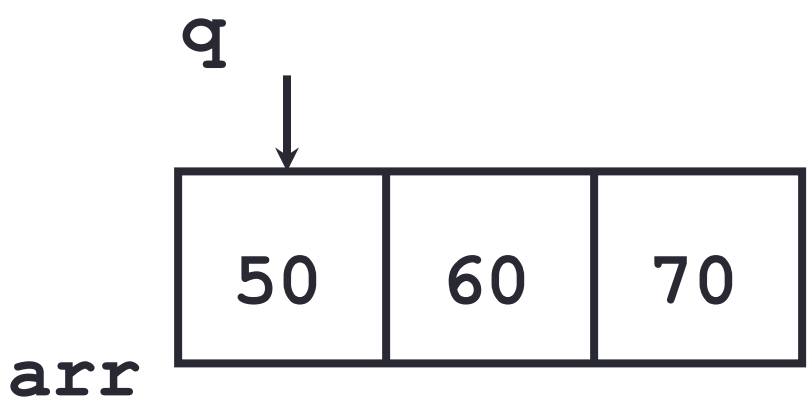
```
void IncrementPtr(int **p){
    p =  p + 1;  }


int arr[3] = {50, 60, 70};
int *q = arr;
IncrementPtr(&q);
```

A. `p =  p + 1;`

B. `&p = &p + 1;`

C. `*p= *p + 1;`

D. `p= &p+1;`



Review of homework 10, problem 5

```
void printRecords(UndergradStudents records [], int numRecords);
int main(){
   UndergradStudents ug[3];
   ug[0] = {"Joe", "Shmoe", "EE", {3.8, 3.3, 3.4, 3.9} };
   ug[1] = {"Macy", "Chen", "CS", {3.9, 3.9, 4.0, 4.0} };
   ug[2] = {"Peter", "Patrick", "ME", {3.8, 3.0, 2.4, 1.9} };
   printRecords(ug, 3);
}
```

Expected output

These are the student records:
ID# 1, Shmoe, Joe, Major: EE, Average GPA: 3.60
ID# 2, Chen, Macy, Major: CS, Average GPA: 3.95
ID# 3, Pan, Patrick, Major: ME, Average GPA: 2.77

Review of homework 10, problem 5

```
void printRecords(UndergradStudents records [], int numRecords)
{
    double avgGPA;
    for(int i=0; i< numRecords; i++){




        cout<< "ID#" <<i <<", " <<records[i].lastName <<", "
        << records[i].firstName << " Avg GPA:" << avgGPA <<endl;
    }
}
```

Review of hw10, P5

```
void printRecords(UndergradStudents *records, int numRecords)
{
    double avgGPA;
    for(int i=0; i< numRecords; i++){
        avgGPA=0;
        for(int j=0; j< NUMGPA; j++){



        }
        cout<< "ID#" <<i <<", " <<records[i].lastName <<", "
        << records[i].firstName << " Avg GPA:" << avgGPA <<endl;
    }
}
```

Pointer Arithmetic

- What if we have an array of large structs (objects)?
  - C++ takes care of it: In reality, `ptr+1` doesn't add `1` to the memory address, but rather adds the size of the array element.
  - C++ knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes: 1 byte for a char, 4 bytes for an int, etc.

Complex declarations in C/C++

How do we decipher declarations of this sort?

```
int **arr[];
```

Read

* as “pointer to” (always on the left of identifier)

[] as “array of” (always to the right of identifier)

( ) as “function returning” (always to the right …)

For more info see:
http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Complex declarations in C/C++

Right-Left Rule

int **arr [];

Illegal combinations include:

[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array

Step 1: Find the identifier

Step 2: Look at the symbols to the right of the identifier. Continue right until you run out of symbols *OR* hit a *right* parenthesis ")"

Step 3: Look at the symbol to the left of the identifier. If it is not one of the symbols '*', '(), '[]' just say it. Otherwise, translate it into English using the table in the previous slide. Keep going left until you run out of symbols *OR* hit a *left* parenthesis "(".

Repeat steps 2 and 3 until you've formed your declaration.

Complex declarations in C/C++

```
int i;
int *i;
int  a[10];
int f( );
int **p;
int (*p)[];
int (*fp) ( );
int *p[];
int af[]( );
int *f();
int fa()[];
int ff()();
int (**ppa)[];
int (*apa[ ])[ ] ;
```

Next time

- Midterm Review
- There will also be a review during next week's section