Function pointer

void (*f)(int);
Named f, takes one parameter which is an int. Won't be on the midterm
f = &squared; (where squared is the name of a function that takes one int)
f(10);

Dynamic Memory Allocation

- Sometimes you won't know how many variables you'll need until your program runs
- Can dynamically ask the operating system to reserve new memory for variables
- Operating system will allocate room for your variable in the computer's free memory and then return the address of the new variable

For an array:
We want to define an array, don't know how big to make it
New command can be used to allocate an arbitrary amount of memory for an array

int *arr;
Int size;
Cin >> size;
arr = new int[size];
delete [] arr; -- don't forget brackets when deleting an array
When you call delete, you are deleting the memory that has been allocated, not the pointer itself

If you have an array of pointers to objects, you need to do a loop to delete every single thing in that array

Copy Construction

- Required in all nontrivial C++ programs
- Define a constructor that accepts another of the same class as a parameter → acceptable
- Every variable of the same type can access the privates of every other of the same type of variable
- You can do Circ b = a; which does the same thing; calls a copy constructor, simpler but ugly
- Otherwise you do Circ(const Circ &old) {}
- Circ b = a is bad. Shallow copy causes problems when you destruct either copy
- Just define your own copy constructor

Assignment Operators

- Will be on the exam
- Required
- If you fail to use them properly, it can result in nasty bugs and crashes
- Change the value of an existing variable to the value of another variable
- Function name is operator=, the return type is a reference
- E.g. Circ &operator=(const Circ &src) {}
- Will call the function if you do bar=foo and call the operator thing and put foo as src

Linked lists

- Super important
- Arrays aren't always great
-

Boelter 2444 Thursday 1/22 6-8
Hackathon on friday

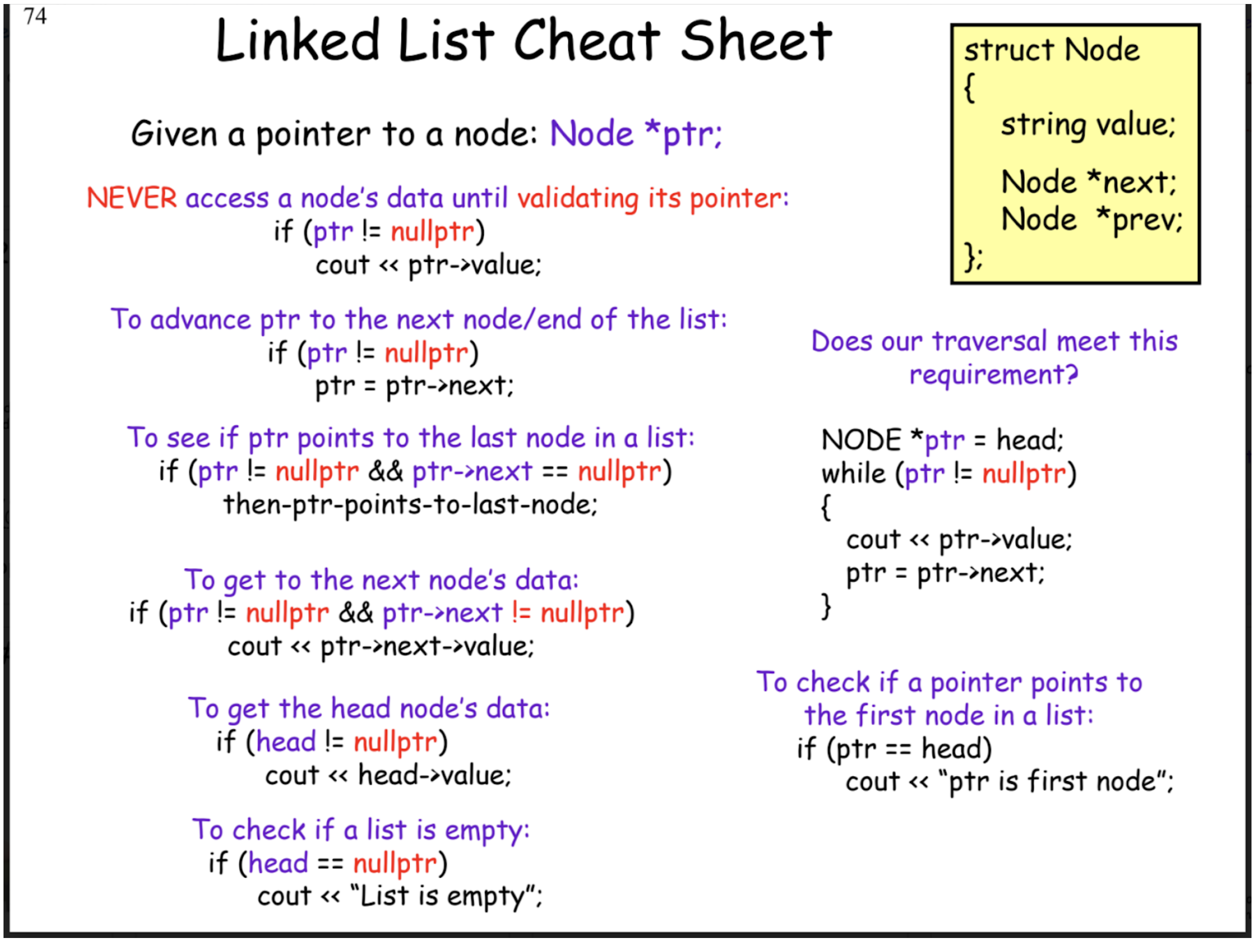
Linked Lists

- Have a head pointer
- That points to the next node
- Which points to the next node
- Etc
- Adding to front
    - Allocate a new node
    - Put value in the node
    - Link the new node to the old top node
        - Node *p;
        - p=new Node;
        - p->value=v;
        - p->next=head
        - head=p
- Adding to rear
    - 2 cases
        - List is totally empty
            - Just add the node
            - If linked list is empty use addToFront code
            - If head==nullptr use addtofront
        - List has stuff

        - Else:
        - Traverse until we find the last node
        - Node *p;
        - p=head;
        - while(p->next != nullptr)
            - p=p->next;
        - Node *n = new Node;
        - n->value=v;
        - p->next=n;
        - n->next=nullptr;
- And more stuff - see slides/handouts

Doubly-linked lists

- Downside of linked list is that you can only look next
- Doubly linked lists have both next and previous



74

Linked List Cheat Sheet

Given a pointer to a node: Node *ptr;

NEVER access a node's data until validating its pointer:

```
if (ptr != nullptr)
    cout << ptr->value;
```

To advance ptr to the next node/end of the list:

```
if (ptr != nullptr)
    ptr = ptr->next;
```

To see if ptr points to the last node in a list:

```
if (ptr != nullptr && ptr->next == nullptr)
    then-ptr-points-to-last-node;
```

To get to the next node's data:

```
if (ptr != nullptr && ptr->next != nullptr)
    cout << ptr->next->value;
```

To get the head node's data:

```
if (head != nullptr)
    cout << head->value;
```

To check if a list is empty:

```
if (head == nullptr)
    cout << "List is empty";
```

```
struct Node
{
   string value;

   Node *next;
   Node  *prev;
};
```

Does our traversal meet this requirement?

```
NODE *ptr = head;
while (ptr != nullptr)
{
   cout << ptr->value;
   ptr = ptr->next;
}
```

To check if a pointer points to the first node in a list:

```
if (ptr == head)
    cout << "ptr is first node";
```