# Data Structures in C

CSC 230 : C and Software Tools

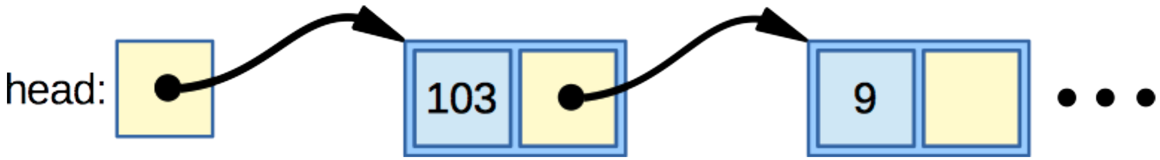
NC State Department of Computer Science

# Topics for Today

- Linked structures in C
- Linked list review
- Methods for maintaining a list
- Object-orientation, simulated
- Deleting a list
- Removing a node
- Meet pointer-to-pointer

# Linked Structures

- Two approaches to keeping data organized:
  - Pack values consecutively, in the same region of memory: an array or a struct.
  - Use pointers to make connections between different data items: a *linked structure*
- A linked list is a basic linked structure
  - With each node holding a value …
  - … and a pointer to the next node.



# Linked Structure Tradeoffs

- Compared to arrays, linked structures offer certain tradeoffs
  - Easier to grow the structure a little bit at a time ☺
  - Don’t need to find large, contiguous regions of memory ☺
  - Can come up with some creative ways to organize the structure, well suited to certain operations ☺
    - Linked lists, balanced trees, disjoint sets, tries …
  - Locality may be worse, jumping all over memory ☹
  - Need to traverse links, instead of using pointer arithmetic to go right where we need to ☹

# Building a Linked List in C

- We can use a struct to represent a node.

```
struct NodeTag {
   int value;
   struct NodeTag *next;
};

typedef struct NodeTag Node;
```

Value in a node.

Pointer to the next node.

A short name to call the node by.

```
Node *head = NULL;
```

A head pointer for the first node on the list.

# Adding to a List

- We can use dynamic memory allocation to get space for each node.

```
Node *n = (Node *)malloc( sizeof( Node ) );

n->value = val;

n->next = head;
head = n;
```

Get space for a node.

Store a value in it.

Link it in at the front.

- We could allocate a bunch of nodes statically
  - but dynamic allocation lets us grow the list as big as we need.

# Traversing a List

- In C, a NULL pointer evaluates to false
- We can use this to write some very simple traversal code

Start at the head.

While the current node is non-NULL

Move to the next node.

```
for ( Node *n = head; n; n = n->next )
  printf( "%d ", n->value );
printf( "\n" );
```

# Organizing our List Code

- We'd like to organize our code
  - Instead of putting linked list code all over the place
  - We'd like organize list operations into reusable functions
- For example:

```
Node *head = NULL;

void addValue( int val )
{
  Node *n = (Node *)malloc( sizeof( Node ) );
  n->value = val;

  n->next = head;
  head = n;
}
```

Here's our list, a global pointer.

A function to add a new value.

Make a node holding the new value.

Link it into the list pointed to by head.

# Organizing our List Code

- This organization has some disadvantages.
  - We store the list as a global
  - So, we can only have one list, the global one.
- To support multiple, independent lists, we'll need to be more creative.

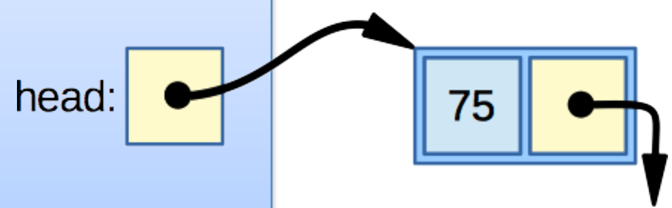
```
Node *head = NULL;

void addValue( int val )
{
  Node *n = (Node *)malloc( sizeof( Node ) );
  n->value = val;

  n->next = head;
  head = n;
}
```

# Organizing our List Code

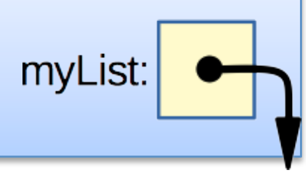
- How about this? We pass in the list as a parameter.

```
void addValue( Node *head, int val )
{
  Node *n = (Node *)malloc( sizeof( Node ) );
  n->value = val;

  n->next = head;
  head = n;
}
```

But, here we have a problem..




```
  Node *myList = NULL;
  …;
  addValue( myList, 75 );
```



# List Passing Alternatives

- One solution, return the new head pointer

```
Node *addValue( Node *head, int val )
{
  Node *n = (Node *)malloc( sizeof( Node ) );
  n->value = val;

  n->next = head;
  return n;
}
```

Here's the new head of the list.

```
  Node *myList = NULL;
  …;
  myList = addValue( myList, 75 );
```

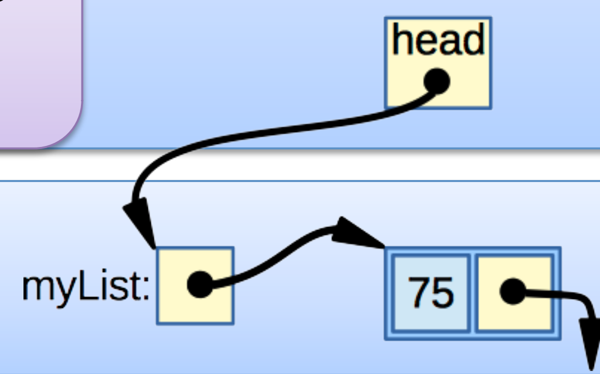
# List Passing Alternatives

- Or, we could pass the head by reference, so the function could change it.
  - That's a pointer to a pointer to a Node

```
void addValue( Node **head, int val )
{
  Node *n = (Node *)malloc( sizeof( Node ) );
  n->value = val;

  n->next = *head;
  *head = n;
}
```

Pointer dereference to get to the caller's head pointer.

```
  Node *myList = NULL;
  …;
  addValue( &myList, 75 );
```



# List Passing Alternatives

- Or, more typically, we could make a little structure representing the list itself.

```
typedef struct {
  Node *head;

} List;
```

I represent a whole list.

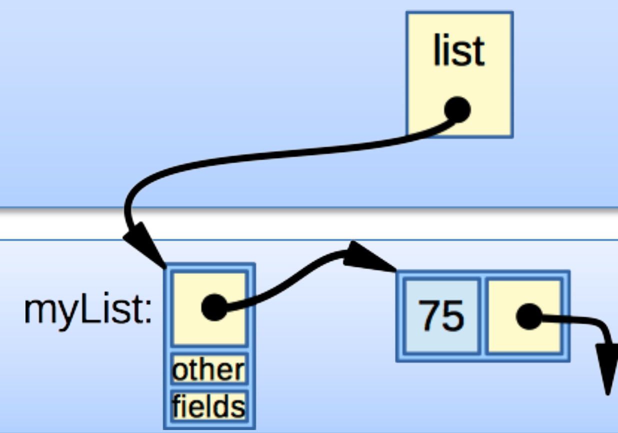
Here's the head pointer.

Plenty of room for more fields if you need them (e.g., a tail pointer)

- We just need to pass a pointer to this struct to every function that uses a list.

# List Passing Alternatives

```
void addValue( List *list, int val )
{
  Node *n = (Node *)malloc( sizeof( Node ) );
  n->value = val;

  n->next = list->head;
  list->head = n;
}
```

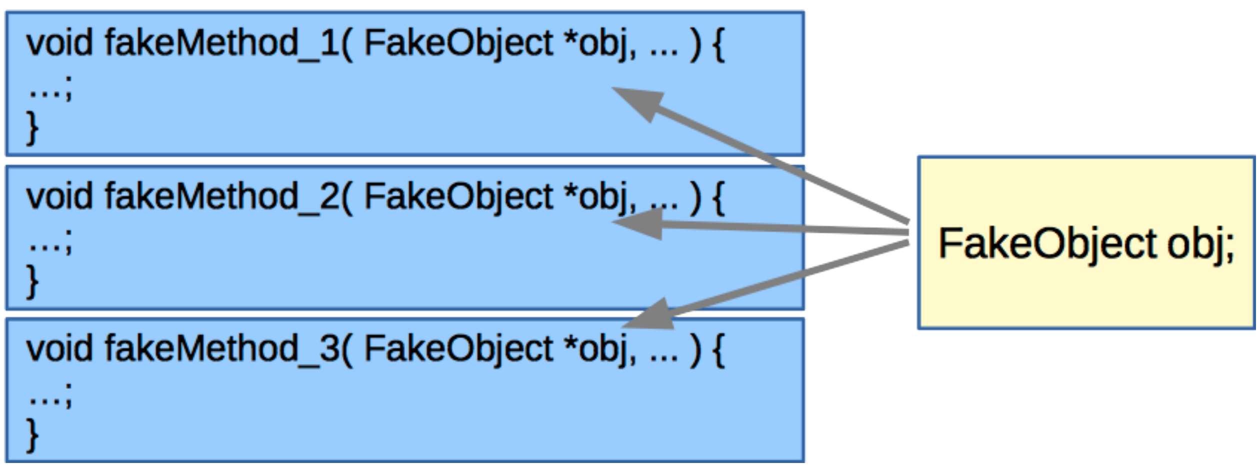



```
  List myList = { NULL };
  …;
  addValue( &myList, 75 );
```

- A pointer to a List struct lets a function:
  - Read fields related to the list
  - Or, modify the list if needed

# Faking Object Orientation

- We could use this to simulate object-orientation
- Instead of member functions we could
  - Define a bunch of (free) functions
  - Pass in a pointer to the object (struct) these functions are supposed to work on.




- Surprise! This is how Java and C++ work internally

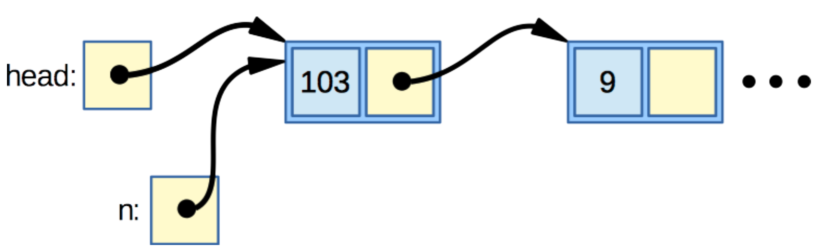
# Freeing Lists

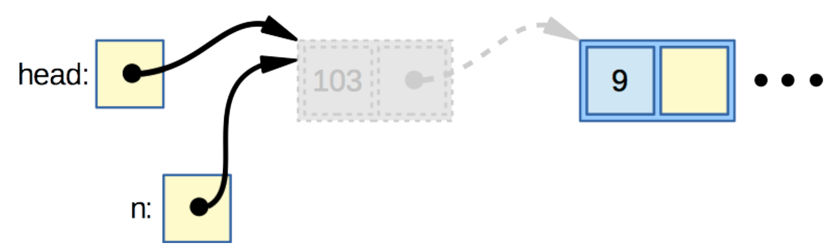
- Eventually, we have to free the memory for our linked list
- Here’s a bad way to do it:

```
for ( Node *n = head; n; n = n->next )
  free( n );
```

Didn’t you just free that node?

- This tries to use a node, right after we free it.



# Freeing Lists … Properly

- This is better:

```
while ( head ) {
  Node *next = head->next;
  free( head );
  head = next;
}
```

Copy the pointer to the next node.

Then free it.

# Getting Creative

- Access to pointers lets us be creative when we write code for linked structures
  - We saw this when we passed a head pointer by reference, so the function could change where it points.
- This comes in handy all over the place … if you know how to use it.
- We'll see pointer-to-pointer today, then in a little more detail later.

# Linked List Representation

- Same representation as before.

```
struct NodeTag {
  int value;
  struct NodeTag *next;
};

typedef struct NodeTag Node;
```

Representation for a Node.

```
typedef struct {
  Node *head;

} List;
```

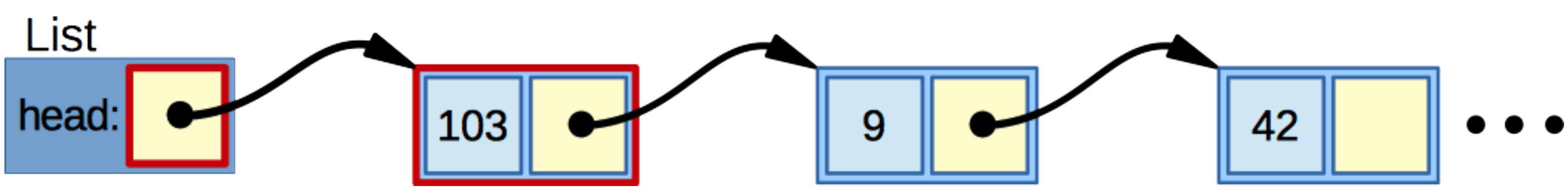
Representation for a whole list.

# Removing Nodes, Classic

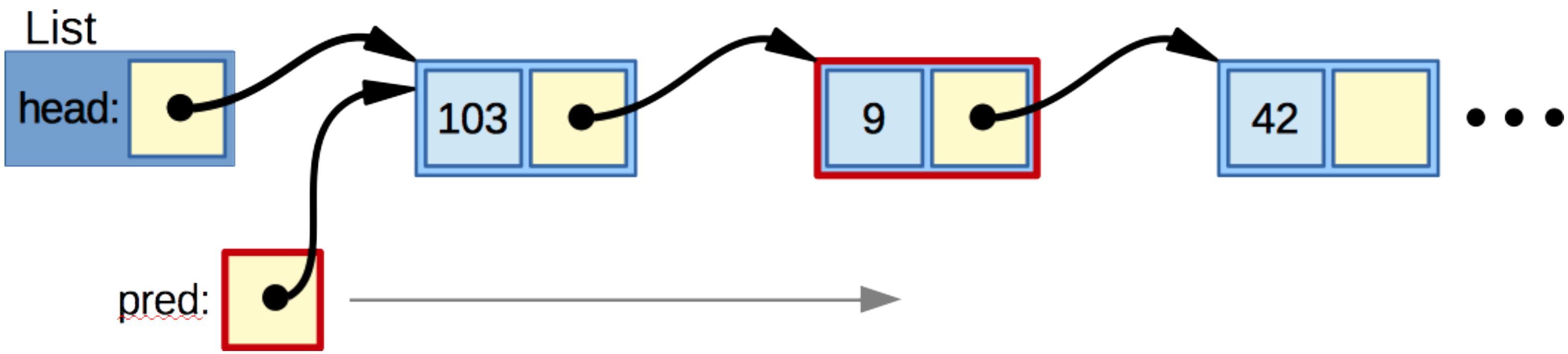
- Normally, to remove a node we might handle the head of the list as a special case:

```
bool removeValue( List *list, int val )
{
  if ( list->head && list->head->value == val ) {
    Node *n = list->head;
    list->head = list->head->next;

    free( n );
    return true;
  }
  …
```

First node as a special case.



# Removing Nodes, Classic




```
…;
Node *pred = list->head;
while ( pred->next && pred->next->value != val )
  pred = pred->next;

if ( pred->next ) {
  Node *n = pred->next;
  pred->next = pred->next->next;

  free( n );
  return true;
}

return false;
}
```

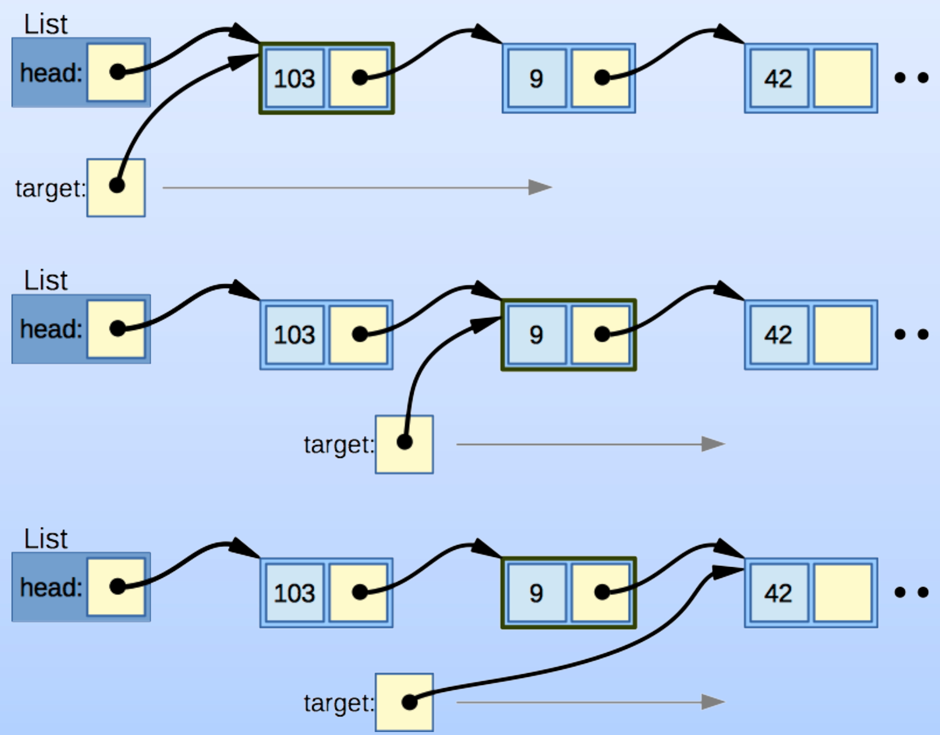
Look for the node **before** the one you want to remove.

Unlink (and free) it.

# An Idea that Almost Works

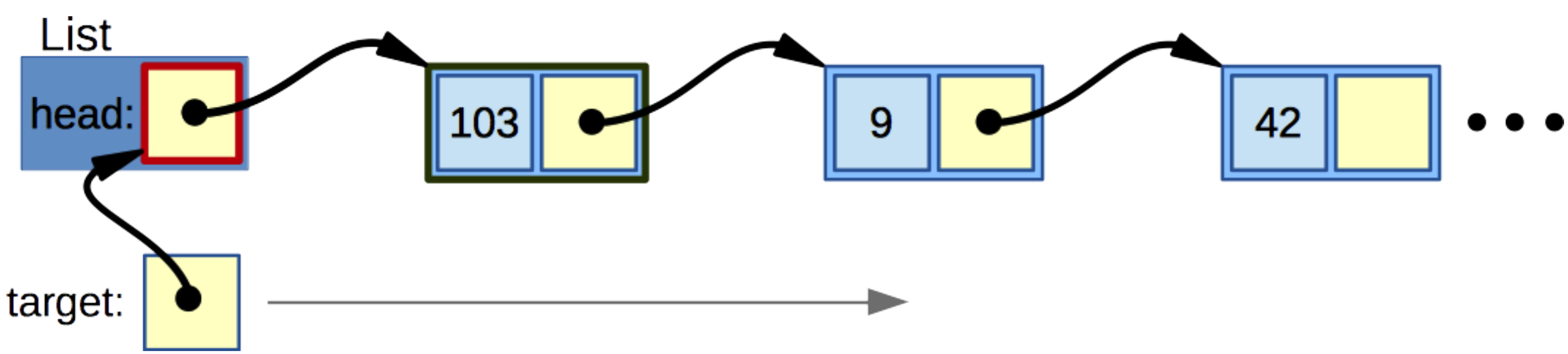
- How about this technique? (it doesn't really work)

```
bool removeValue( List *list, int val )
{
  Node *target = list->head;
  while (target &&
         target->value != val)
    target = target->next;

  if ( target ) {
    Node *n = target;
    target = target->next;

    free( n );
    return true;
  }


  return false;
}
```



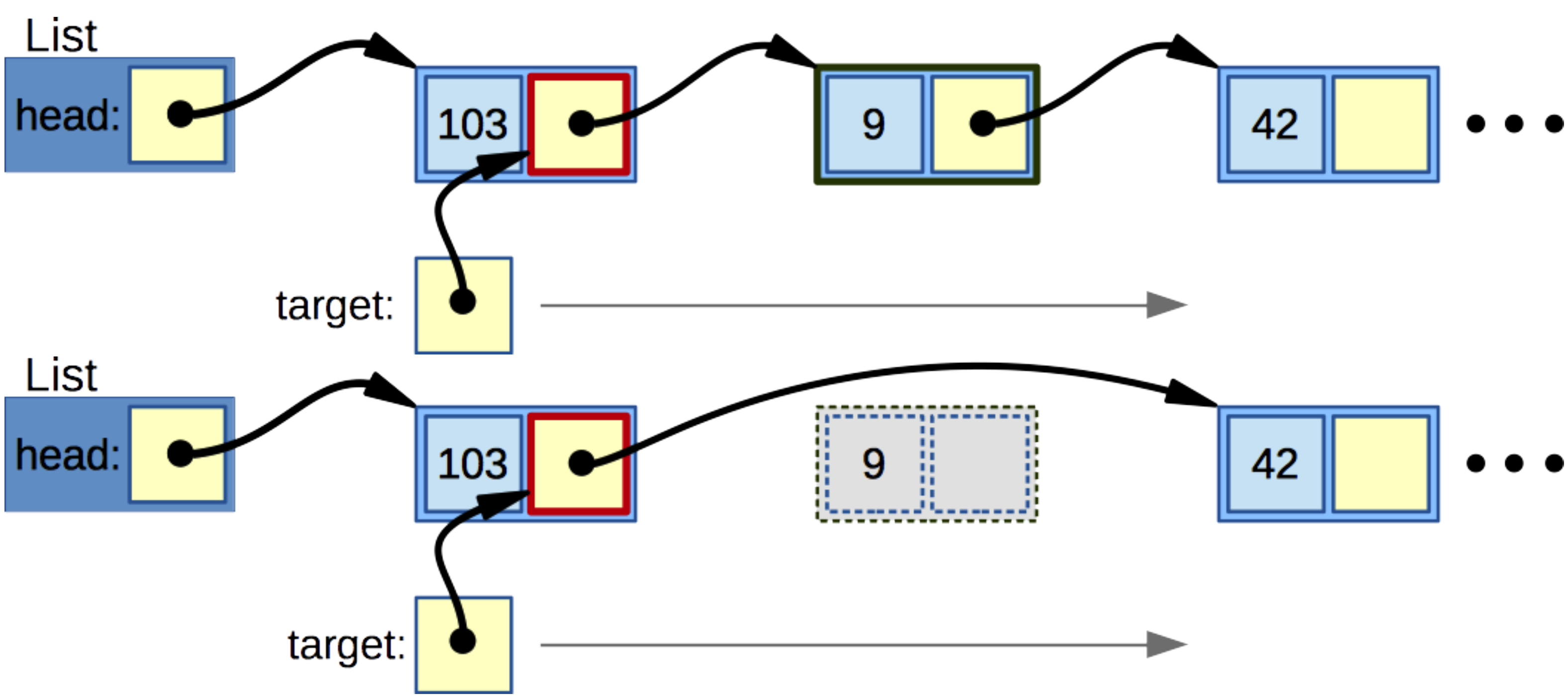
# Removing Nodes, "Simplified"

- To remove a node, we must change the pointer that points to it.
  - Every node has such a pointer
  - Either **the pointer inside its predecessor**.
  - **Or, the head pointer** itself.
- Both of these pointer have the same type (pointer to Node)
- We can handle them uniformly via a pointer to pointer to Node.
- We'll start with a pointer pointing to the head pointer:



# Removing Nodes, “Simplified”

- As we traverse the list, we will move the ahead to pointers within the nodes.
  - This will give us a way to remove the next node on the list.



# Removing Nodes, Simplified

```
bool removeValue( List *list, int val )
{
  Node **target = &list->head;
  while ( *target && (*target)->value != val )
    target = &(*target)->next;

  if ( *target ) {
    Node *n = (*target);
    *target = (*target)->next;

    free( n );
    return true;
  }

  return false;
}
```

I'm the address of the pointer to the node you're thinking about removing.

Start at the head, then walk down through all the pointers inside the nodes.

If we find a node containing val, remove its node by changing the pointer that points to it.