Linked List

- Define a *node* class (or struct) with two data members:
  - One to store a piece of data
  - One that is a pointer to another *node* object

```
class node
{
public:
    int data;
    node * next;

    // constructor for convenience
    node( int value ) { data = value; }
};
```

Creating a List

- Using the *node* class, we can create a *linked list* to hold data
  - E.g. the integers 35, 65, 34, 76
  - Like an array, but with very important differences

Creating a List

- Declare a *node* pointer called *head*:

```
node * head = NULL;
```

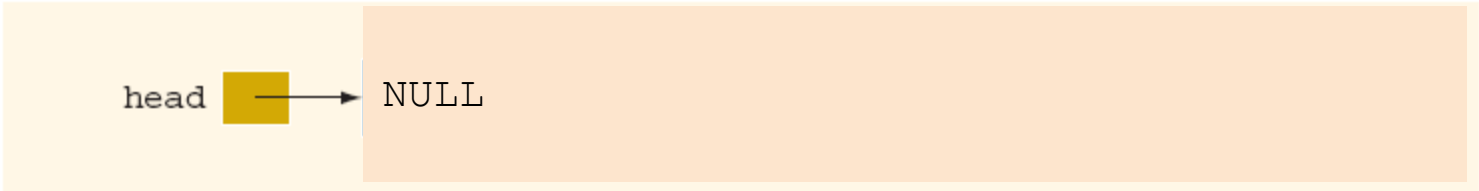

FIGURE 17-2 Linked list

Linked List

- Make head point at a new *node* object holding the first value

```
node * head = NULL;
head = new node( 45 );
```

FIGURE 17-2 Linked list

- Note that the list still has a pointer to NULL at the end

Creating a List

- Add another new *node* object for the next value

```
node * head = NULL;
head = new node( 45 );
head->next = new node( 65 );
```

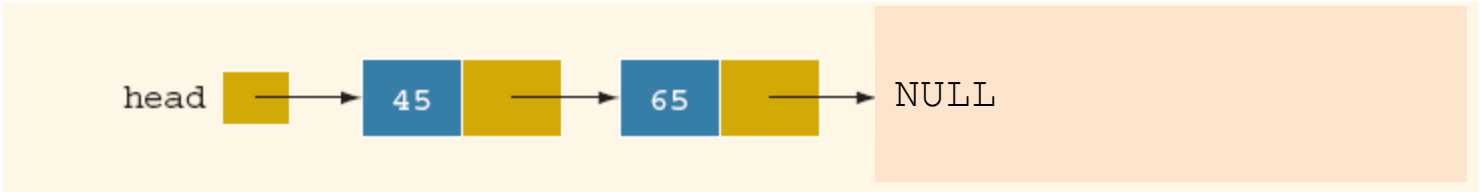

FIGURE 17-2 Linked list

- Note that the list still has a pointer to NULL at the end

Creating a List

- And repeat…

```
node * head = NULL;
head = new node( 45 );
head->next = new node( 65 );
head->next->next = new node( 34 );
```

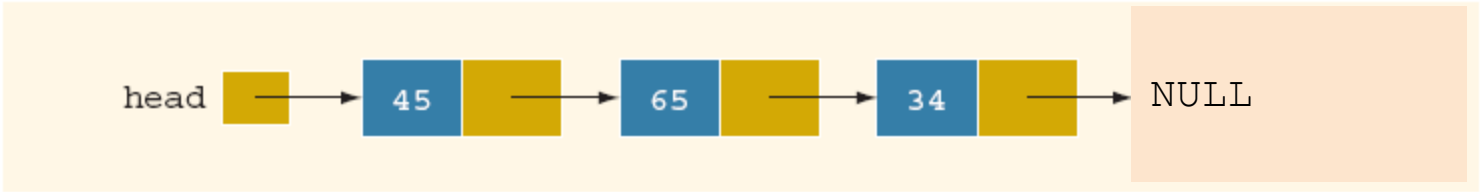

FIGURE 17-2 Linked list

Creating a List

- And repeat…

```
node * head = NULL;
head = new node( 45 );
head->next = new node( 65 );
head->next->next = new node( 34 );
head->next->next->next = new node( 76 );
```

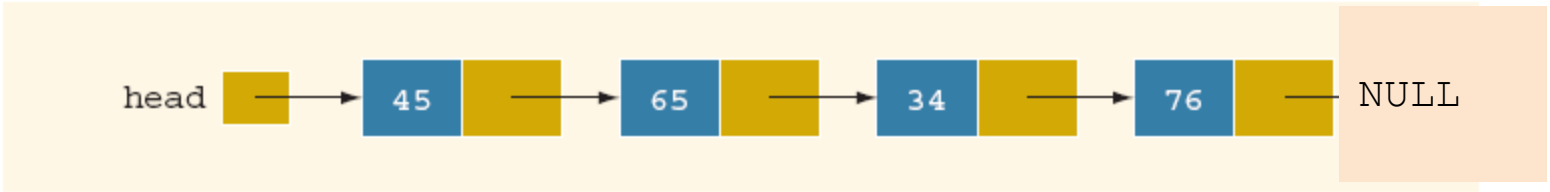

FIGURE 17-2 Linked list

Linked List

- Holds a variable amount of data
  - Like an array, but...
  - Always exactly the right size (no wasted space)
  - Doesn't require copying all the values to resize
  - Doesn't require copying all the values to insert/delete
    - We'll get back to this is a little bit
  - But, doesn't allow *random access*

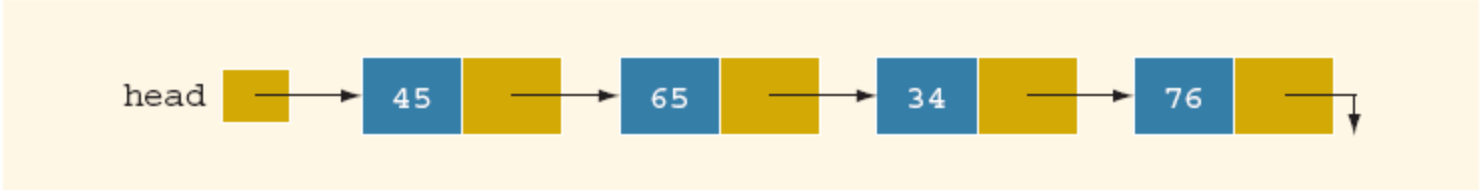

FIGURE 17-2 Linked list

Random Access

- You can access any element in an array in one operation
  - First element? a[0]
  - Last element? a[size]
  - Thirtieth element? a[29]
  - Middle element? a[size/2]
- Linked lists only support *sequential access*
  - First element? head->data
  - Second element? head->next->data
  - Third element? head->next->next->data

Iteration

- Many array operations involve iterating over the elements of the array
  - For loop!
  - This is sequential access
- All linked list operations involve iterating over the elements by *traversing* (stepping through) them

Traversal

- Given a list with the pointer head
- Create a temporary pointer to traverse the list
  - Head is only used to hold the start of the list!

```
node * current = head;
cout << current->data << endl;  // print 45
```

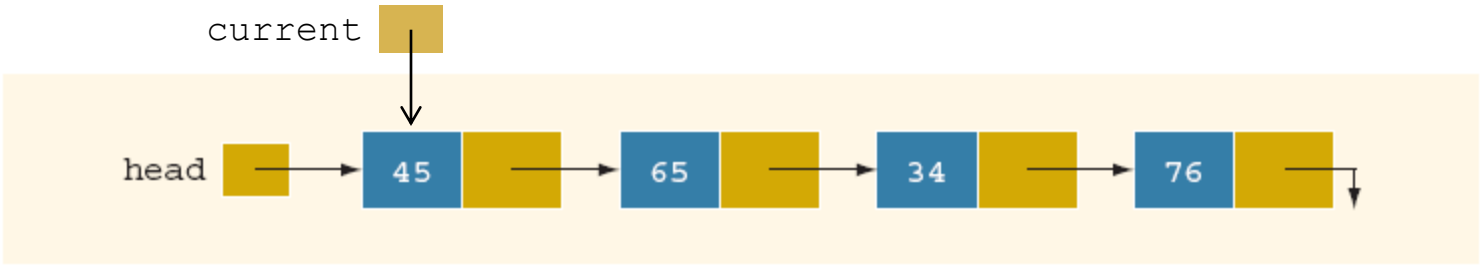

FIGURE 17-2 Linked list

Traversal

- Move the temporary pointer one step forward

```
current = current->next;
cout << current->data << endl ;  // print 65
```

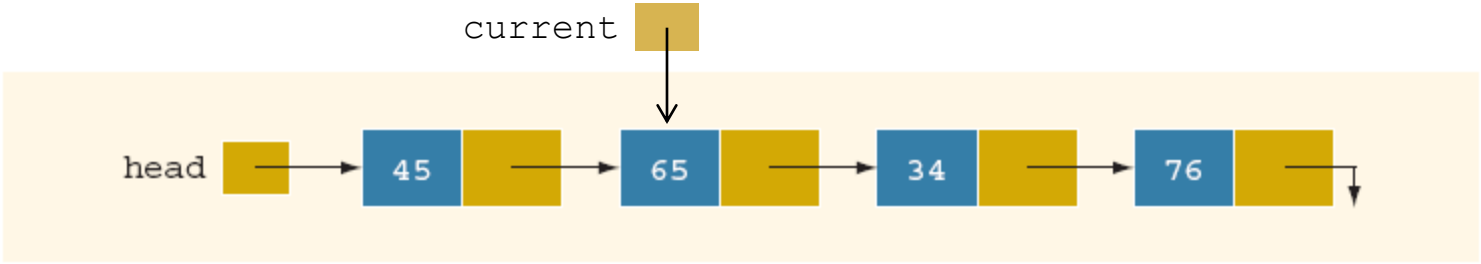

FIGURE 17-2 Linked list

Traversal

- Move the temporary pointer one step forward

```
current = current->next;
cout << current->data << endl ;  // print 34
```

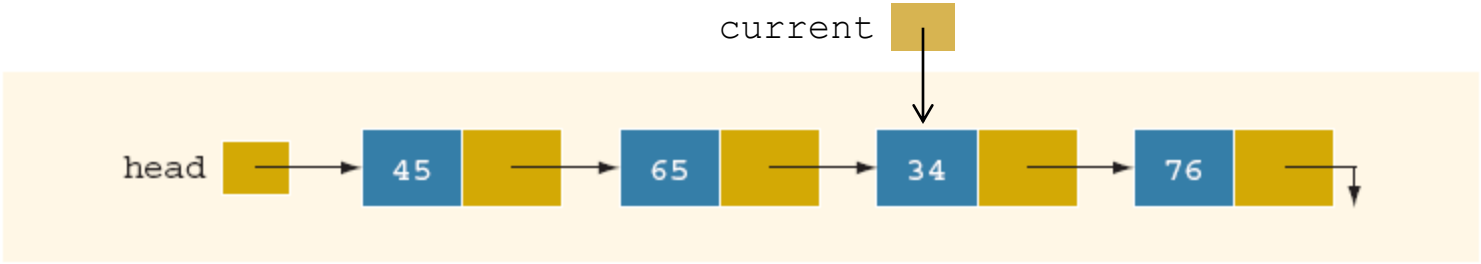

FIGURE 17-2 Linked list

Traversal

- Move the temporary pointer one step forward

```
current = current->next;
cout << current->data << endl ;  // print 76
```

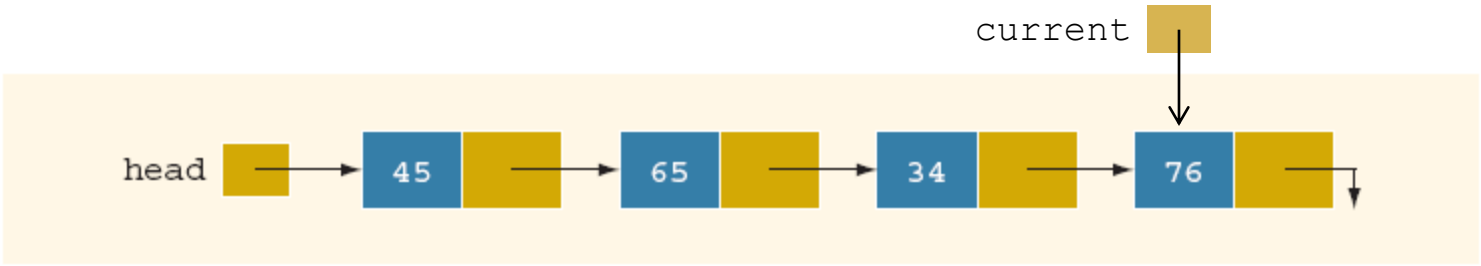

FIGURE 17-2 Linked list

Traversal

- Notice that we just called the same code over and over
  - Should be a loop!

```
node * current = head;
while( ??? ) {
   cout << current->data << endl;
   current = current->next;
}
```

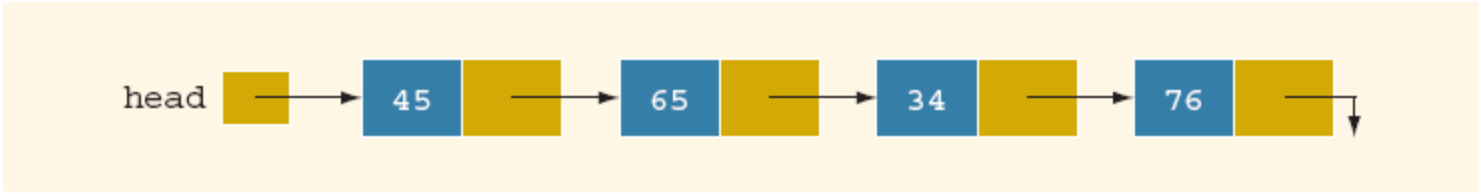

FIGURE 17-2 Linked list

Traversal

- When do we stop?
  - What condition can we use to detect that we are at the end of the loop?

```
node * current = head;
while( ??? ) {
   cout << current->data << endl;
   current = current->next;
}
```

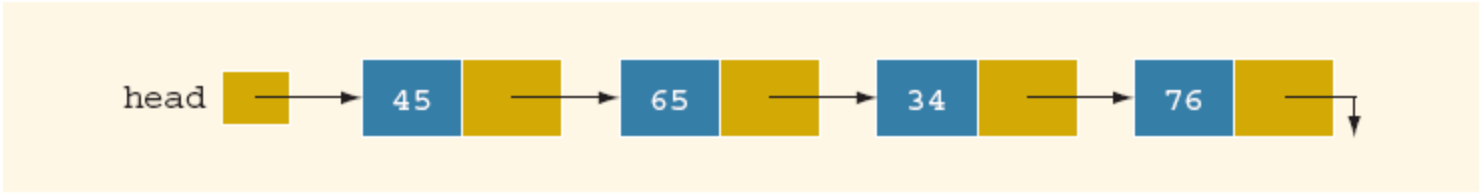

FIGURE 17-2 Linked list

Traversal

- The list always ends with a NULL pointer
  - Loop until current->next == NULL

```
node * current = head;
while( current->next != NULL ) {
   cout << current->data << endl;
   current = current->next;
}
```

  - Trace that code on paper and see if it will work
  - Where does current end up?

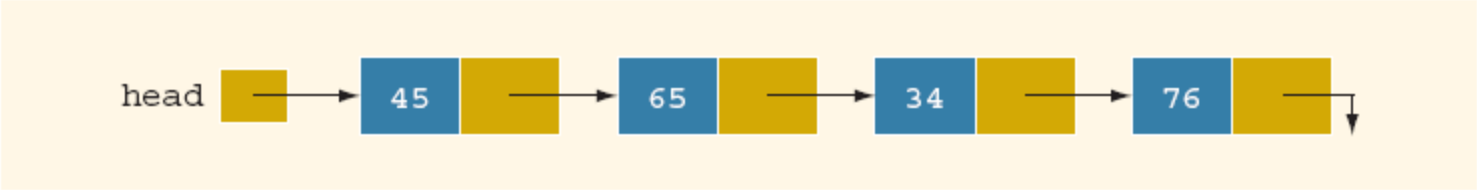

FIGURE 17-2 Linked list

Traversal

- The list always ends with a NULL pointer
  - Loop until current->next == NULL

```
node * current = head;
while( current->next != NULL ) {
   cout << current->data << endl;
   current = current->next;
}
```

  - Trace that code on paper and see if it will work
  - Where does current end up?

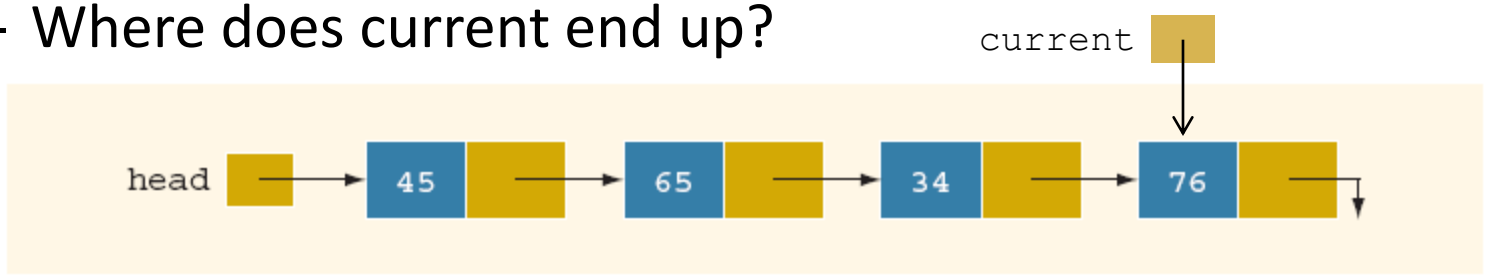

FIGURE 17-2 Linked list

Traversal

- The list always ends with a NULL pointer
  - How about looping until current == NULL?

```
node * current = head;
while( current != NULL ) {
   cout << current->data << endl;
   current = current->next;
}
```

  - Trace that code on paper and see if it will work
  - Where does current end up?

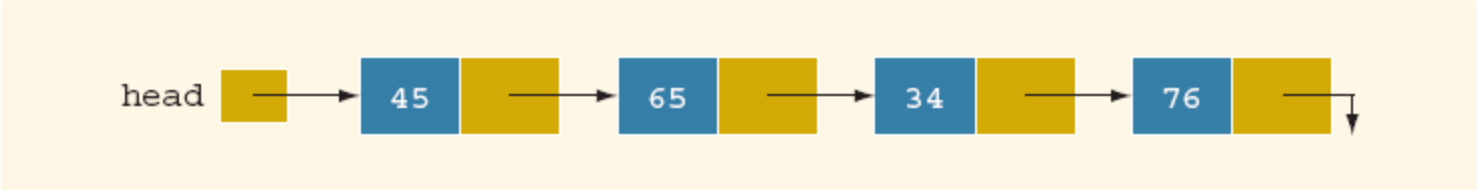

FIGURE 17-2 Linked list

Traversal

- The list always ends with a NULL pointer
  - How about looping until current == NULL?

```
node * current = head;
while( current != NULL ) {
   cout << current->data << endl;
   current = current->next;
}
```

  - Trace that code on paper and see if it will work
  - Where does current end up?

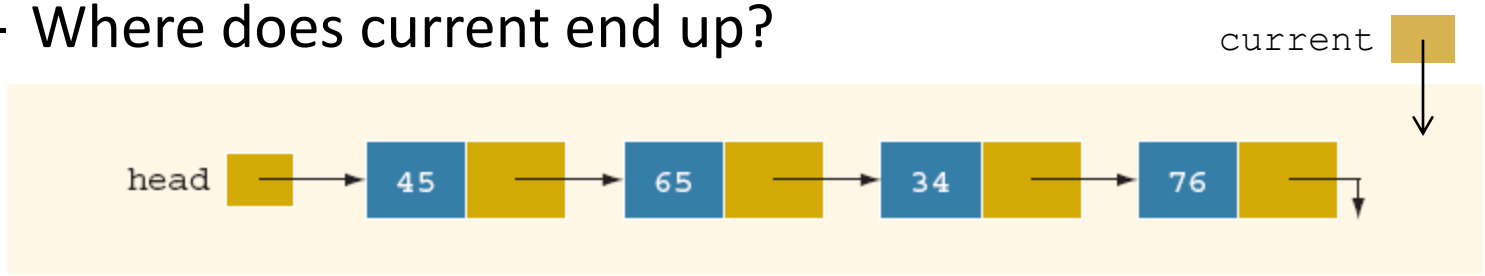

FIGURE 17-2 Linked list

Traversal

- Linked lists must support a number of functions:
  - Size, find, append, insert, delete, etc...

- They all involve traversal of some sort:

```
node * current = head;
while( <check some condition on current> ) {
  <do something for each element>
  current = current->next;
}
<and/or do something at a certain element>
```

  - The key here is to understand how to get to a certain element in the list

Example: print

```
// a function to print a linked list
void print_list( node * head )
{
   node * current = head;
   while( current != NULL ) {
       cout << current->data << " ";
   }
   cout << endl;
}
```

Exercise: find

- Write a function to return true if a value is in a linked list, false otherwise
  - Start with the traversal template:

```
node * current = head;
while( <check some condition on current> ) {
  <do something for each element>
  current = current->next;
}
<and/or do something at a certain element>
```

  - At what element do we stop?
  - What condition can identify that stopping point?
  - Is there something to do at each node?
  - Is there something to do when we stop?