ECE 250 Algorithms and Data Structures

# Lab 1: Linked Lists

Douglas Wilhelm Harder and Hanan Ayad
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

# Outline

- List and Node Classes
- The Constructor
- Simple Function Definitions
- Push Front Definition
- Pop Front Definition
- Stepping through Linked Lists
- The Destructor
- Assignment and Copy Constructor
- Summary

# List and Node Classes

- A linked list is a container where each object is stored in separate *node*
- Each node must, in addition to storing an object, must also include a reference/pointer to the next node
  - In C++, this is the address of the next node

# List and Node Classes

- We must dynamically create the nodes in a linked list
- Thus, because **`new`** returns a pointer, the logical manner in which to track a linked lists is through a pointer
- A **`Node`** class must store the data object and a reference to the next node (also a pointer)

# The Node Class

- The node must store an object and a pointer:
  - In this case, the object is an integer

```
class Node {
    private:
        int   element;
        Node *next_node;
    public:
        Node( int = 0, Node * = 0 );

        int retrieve() const;
        Node *next() const;
};
```

  - In Project 1, the type is given by the template:

```
        Object element;
```

# The Node Class

- The constructor assigns the two member variables based on the arguments

```
Node::Node( int e, Node *n ):element( e ), next_node( n ) {
    // empty constructor
}
```

- The default values for both arguments are both `0`

# The Node Class

- The two member functions are accessors which simply return the **`element`** and the **`next_node`** member variables, respectively

```
int Node::retrieve() const {
     return element;
}

Node *Node::next() const {
    return next_node;
}
```

# Project 1 – Creating your VS Project

- Create a VS project name P1( type is c++, General)
- Add source code provided for P1, in course web page.
  - Start by solving part a) – Single List
  - Files needed in the VS project :
    - Ece250.h
    - Exception.h
    - Tester.h

From Common Source Files

    - Single_list.h
    - Single_node.h
    - Single_node_tester.h
    - Single_list_tester.h
    - Single_list_driver.cpp

# Project 1 – Creating your VS Project

- Compile your VS Project as given. It should compile without errors.
- Complete class Single_node.h
- Questions about C++ syntax so far?
  - Template classes
  - Friend classes
  - Constructor , initializing values

# The List Class

- Because each node in a linked list refers to the next, the linked list class needs to store the address of the first node
  - This requires one member variable: a pointer to a node

```
class List {
    private:
        Node *list_head;
    // ...
};
```

# The List Class

- This is the class definition:

```
class List {
    private:
        Node *list_head;
    public:
        List();

        // Accessors
        bool empty() const;
        int front() const;
        Node * head() const;
        int count( int ) const;

        // Mutators
        void push_front( int );
        int pop_front();
        int erase( int );
};
```

# The List Class

- This is the class in your project 1 – very similar but not the same:

```
template <typename Object>
class Single_list {
private:
   Single_node<Object> *list_head;
   Single_node<Object> *list_tail;
   int node_count;
```

# Linked Lists in Memory

- To begin, let us look at the internal representation of a linked list
- Suppose we want a linked list to store the values

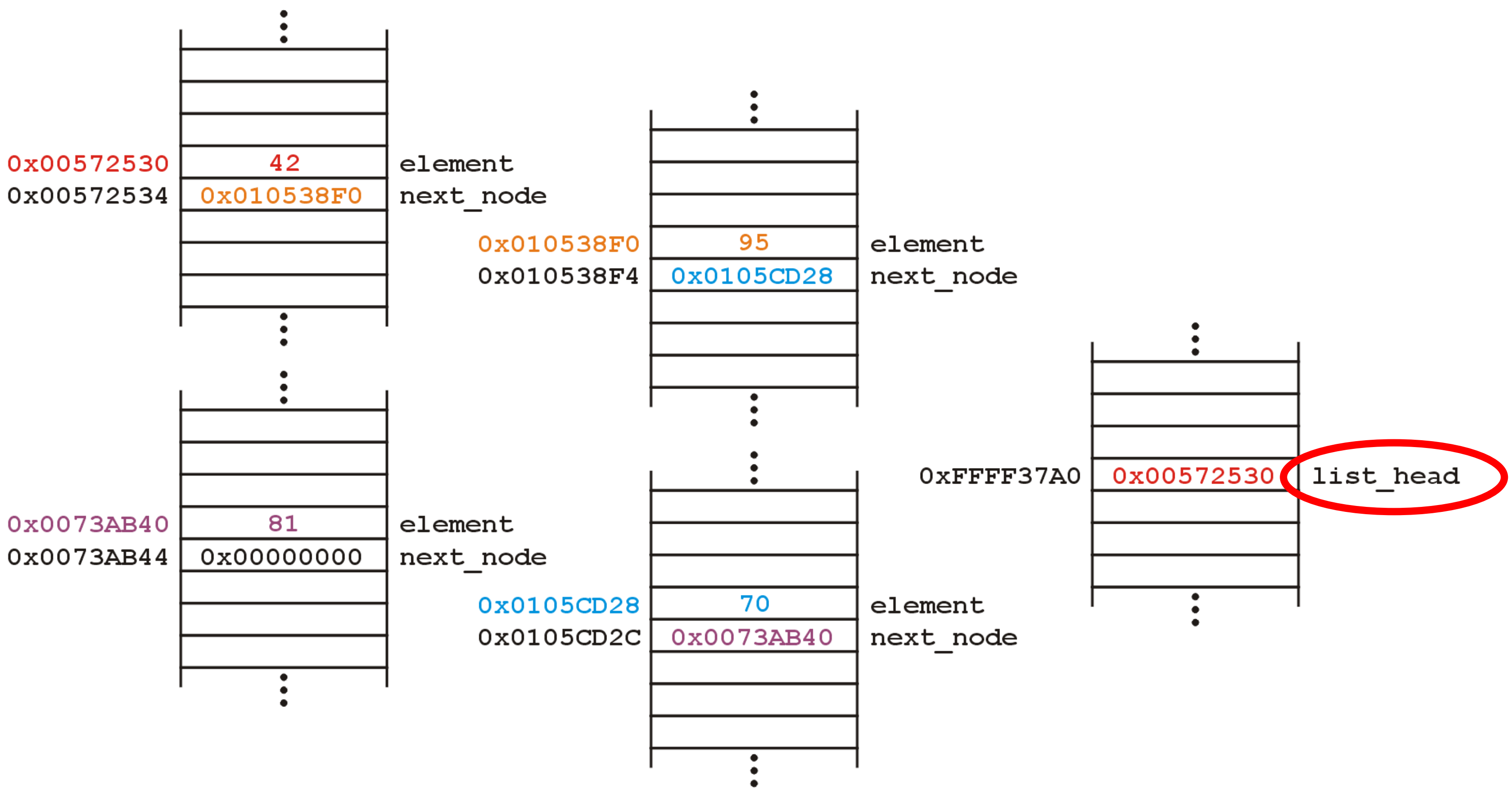
  **42 95 70 81**

  in this order
- This could be the result of:

```
int main() {
    List ls;
    ls.push_front( 81 );
    ls.push_front( 70 );
    ls.push_front( 95 );
    ls.push_front( 42 );
    return 0;
}
```

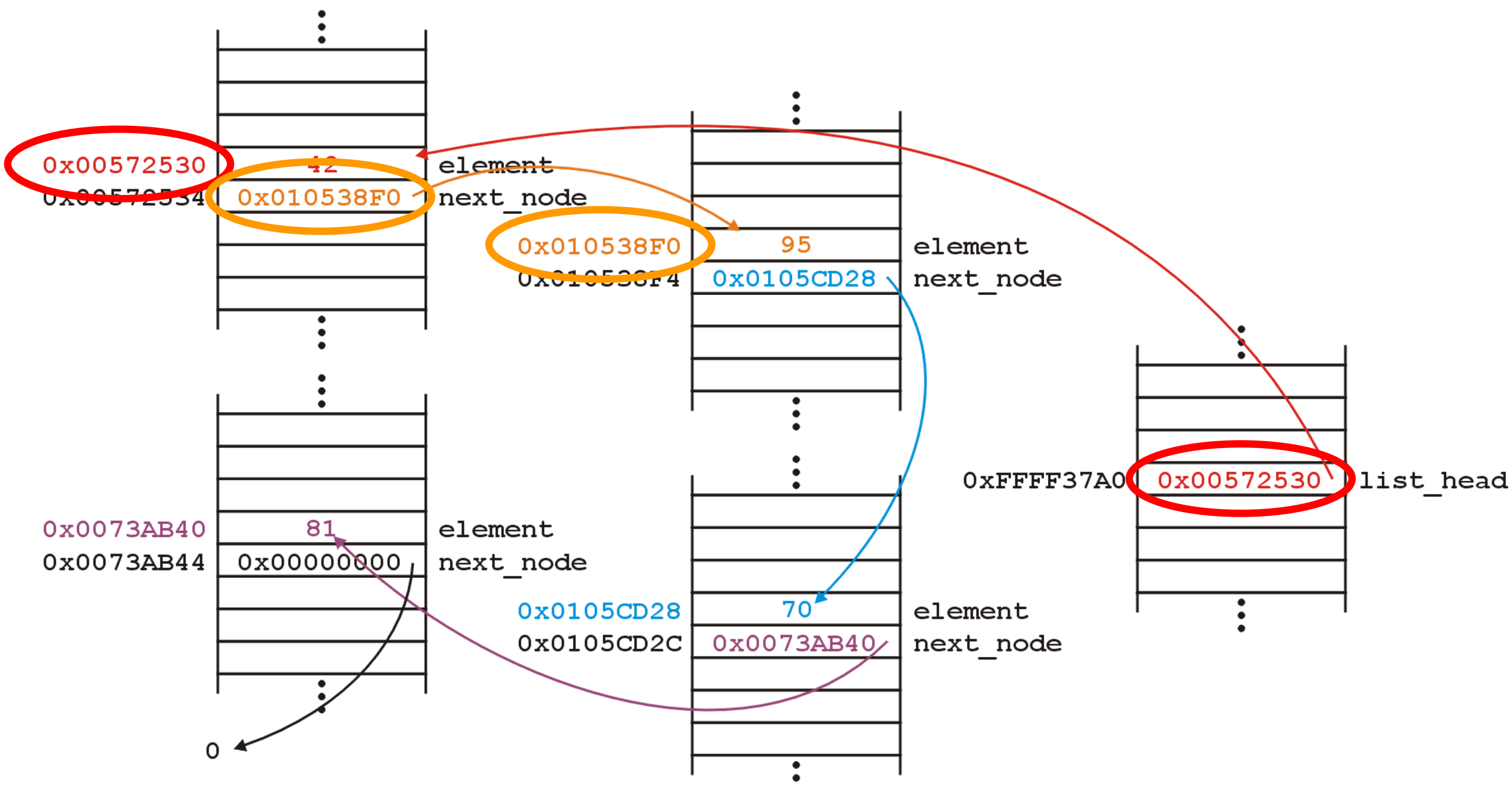
# Linked Lists in Memory

- These nodes could be stored anywhere in memory:
  - For example,



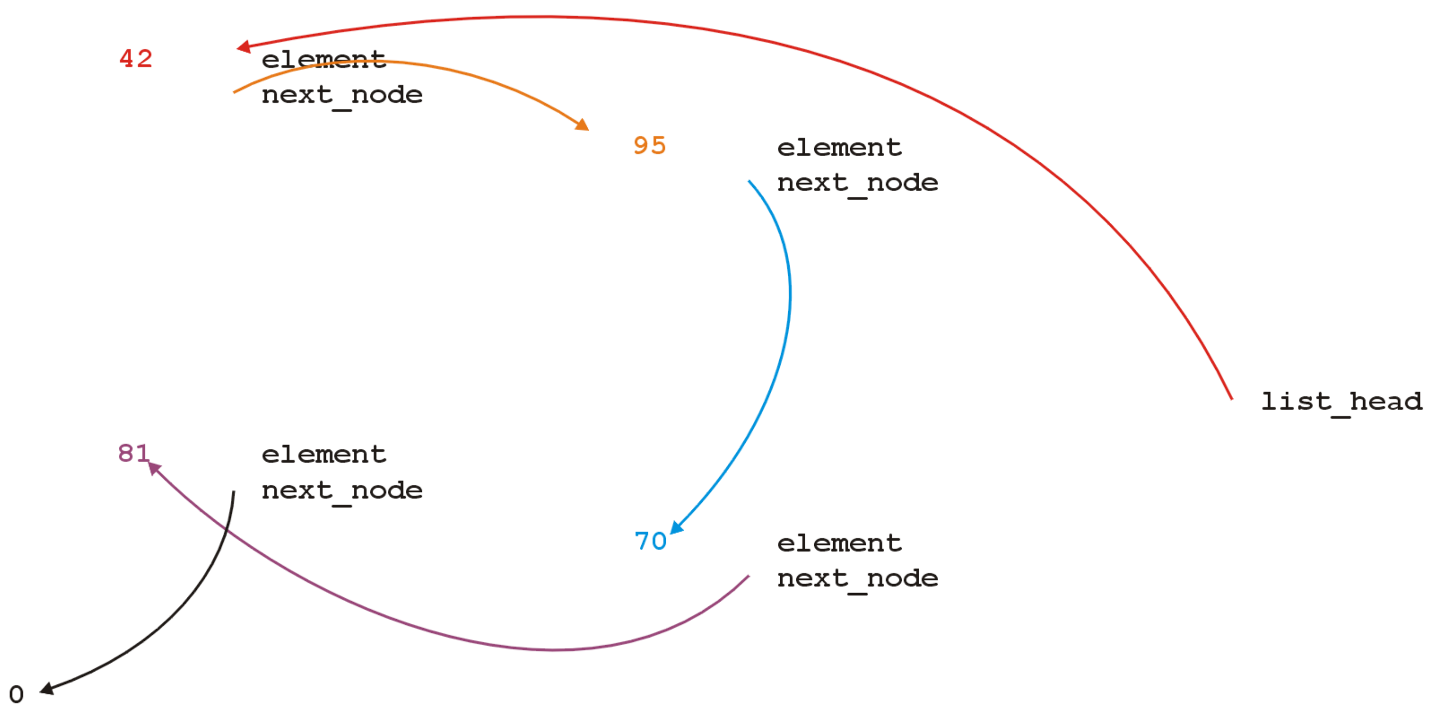
# Linked Lists in Memory

- Notice how each **`next_node`** address points to the next node in the list?



# Linked Lists in Memory

- Because the addresses are arbitrary, we can remove that information:



# Linked Lists in Memory

- We will clean up the representation as follows:


- We do not specify the addresses because they are arbitrary and:
  - The contents of the circle is the stored object
  - The **`next_node`** pointer is represented by an arrow

# Operations on Linked Lists

- First, we want to create a linked list
- We also want to be able to:
  - Insert `void push_front( int );`
  - Access `int front() const;`
    `Node *head() const;`
  - Remove `int pop_front();`

  the front object stored in the linked list
- Operations on the entire linked list may include:
  - Membership `int count( int ) const;`
  - Erase an object `int erase( int );`
- Additionally, we may wish to check the state: is the linked list empty?

  `bool empty() const;`

# The Constructor

- The constructor initializes the linked list
- We do not count how may objects are in this list, thus:
  - We must rely on the last pointer in the linked list to point to a special value
  - In C++, we use **`0`** (the zero address)

# The Constructor

- Thus, in the constructor, we assign **`0`** to **`list_head`**:

```
List::List():list_head( 0 ) {
    // empty constructor
}
```

- We will always ensure that when a linked list is empty, the list head is assigned **`0`**

# Simple Function Definitions

- Starting with the easier member functions:

```
bool List::empty() const {
    if ( list_head == 0 ) {
        return true;
    } else {
        return false;
    }
}
```

or better yet:

```
bool List::empty() const {
    return ( list_head == 0 );
}
```

# Simple Function Definitions

- The member function **`Node *head() const`** is easy enough to implement:

```
Node *List::head() const {
    return list_head;
}
```

- This will always work: if the list is empty, it will return **`0`**

# Simple Function Definitions

- To get the first element in the linked list, we must access the node to which the `list_head` is pointing
- Because we have a pointer, we must use the `->` operator to call the member function:

```
int List::front() const {
    return head()->retrieve();
}
```

# Simple Function Definitions

- The member function **`int front() const`** requires some additional consideration, however:
  - What if the list is empty?
- If we tried to access a member function of a pointer set to **`0`**, we would access restricted memory
- The operating system would terminate the running program

# Simple Function Definitions

- Instead, we can use an exception handling mechanism where we throw an exception
- We define a class

```
class underflow {
    // emtpy
};
```

and then we *throw* an instance of this class:

```
throw underflow();
```

# Simple Function Definitions

- Thus, the full function is

```
int List::front() const {
    if ( empty() ) {
        throw underflow();
    }

    return head()->retrieve();
}
```

# Simple Function Definitions

- Why is `emtpy()` better than

```
int List::front() const {
    if ( list_head == 0 ) {
        throw underflow();
     }

    return head()->retrieve();
}
```

  ?

- Two benefits:
  - more readable
  - if the implementation changes we do nothing

# Push Front Definition

- Next, let us add an element to the list
- If it is empty, we start with:


and, if we try to add 81, we should end up with:

# Push Front Definition

- To visualize what we must do:
  - We must create a new node which stores:
    - The value `81`, and
    - The address (pointer) `0`
  - We must then assign its address to **`list_head`**
- We can do this as follows:

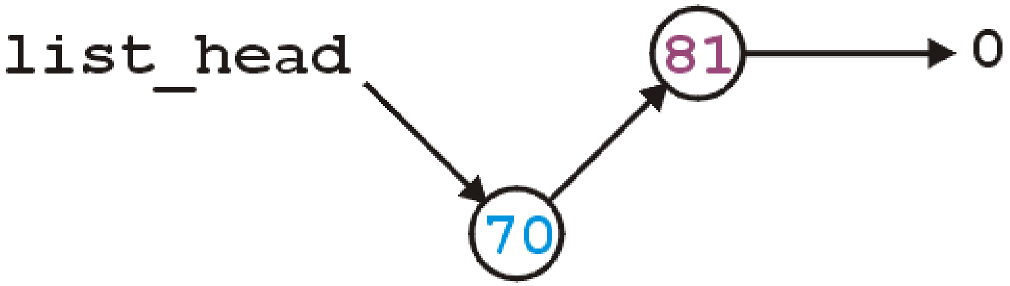
```
list_head = new Node( 81, 0 );
```

- We could also use the default value...

# Push Front Definition

- Suppose however, we already have a non-empty list


- Adding `70`, we want:



# Push Front Definition

- To achieve this, we must we must create a new node which stores:
    - The value **70**, and
    - The address of the current list head
  - We must then assign its address to **list_head**
- We can do this as follows:

```
list_head = new Node( 70, list_head );
```

# Push Front Definition

- Thus, our implementation could be:

```
void List::push_front( int n ) {
    if ( empty() ) {
        list_head = new Node( n, 0 );
    } else {
        list_head = new Node( n, list_head );
    }
}
```

# Push Front Definition

- We could, however, note that when the list is empty, **`list_head == 0`**, thus we could shorten this to:

```
void List::push_front( int n ) {
    list_head = new Node( n, list_head );
}
```

# Push Front Definition

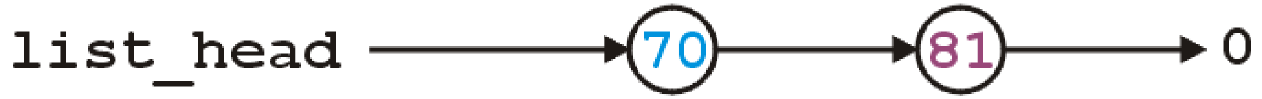
- Are we allowed to do this?

```
void List::push_front( int n ) {
    list_head = new Node( n, list_head );
}
```

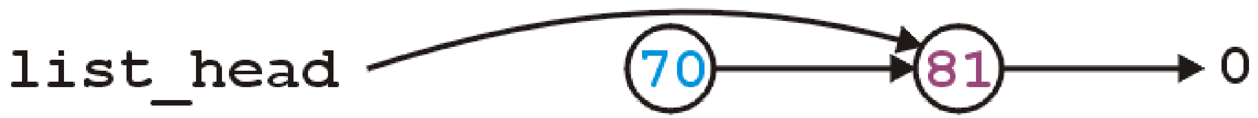
- Yes: the right-hand side of an assignment statement is evaluated first
- The original value of `list_head` is accessed first before the function call is made

# Pop Front Definition

- Removing from the linked list may appear easier:
  - We simply assign the list head to the next pointer of the first node
- Graphically, given:




we want:



# Pop Front Definition

- Easy enough:

```
int List::pop_front() {
    list_head = list_head->next();
}
```

- Unfortunately, we have some problems:
  - We want to return the item stored
  - The list may be empty
  - We still have the memory allocated for the node containing **70**

# Pop Front Definition

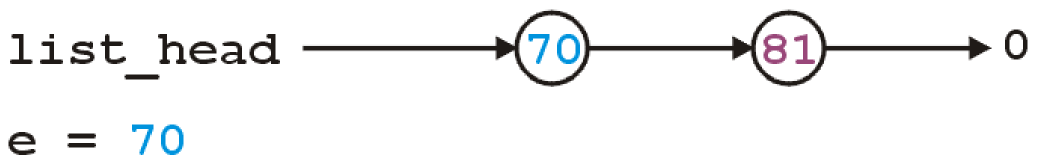
- We could try:

```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    int e = front();
    delete list_head;
    list_head = list_head -> next();
    return e;
}
```

- Let’s examine this...

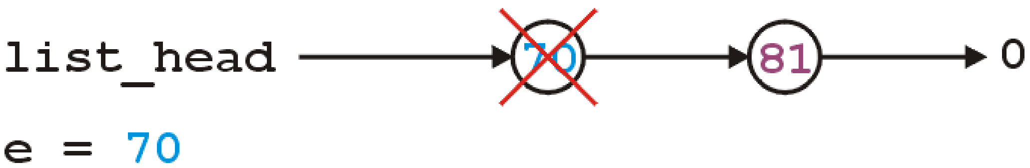
# Pop Front Definition

```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    int e = front();

    delete list_head;

    list_head = list_head->next();

    return e;
}
```



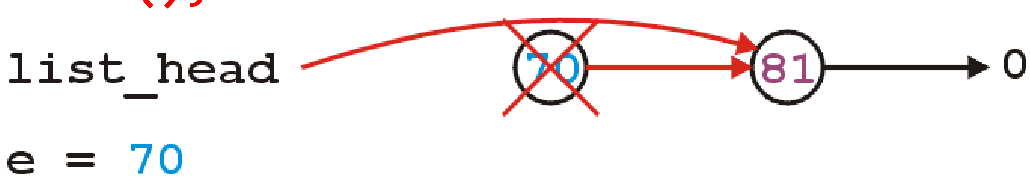
# Pop Front Definition

```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    int e = front();

    delete list_head;

    list_head = list_head->next();

    return e;
}
```



# Pop Front Definition

```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    int e = front();

    delete list_head;

    list_head = list_head->next();

    return e;
}
```



# Pop Front Definition

- The problem is, we are accessing a node which we have just deleted
- Unfortunately, this will work more than 99% of the time:
  - the running program (process) may still own the memory
- Once in a while it will fail ...

... and it will be almost impossible to debug

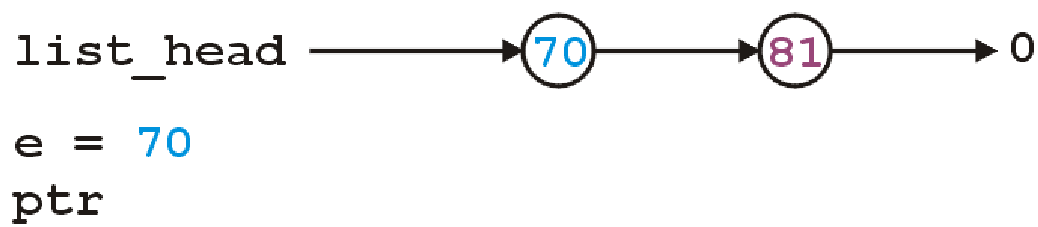
# Pop Front Definition

- The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();
    Node *ptr = list_head;
    list_head = list_head->next();
    delete ptr;
    return e;
}
```

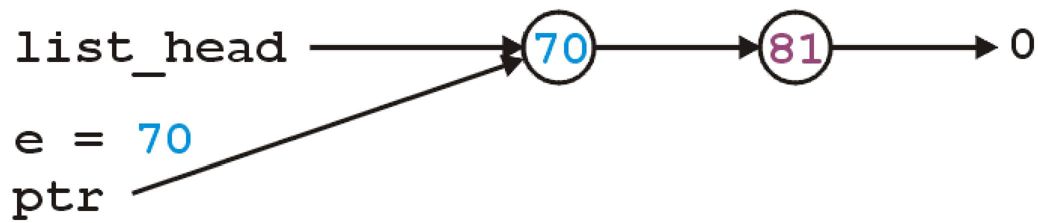
# Pop Front Definition

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();

    Node *ptr = list_head;

    list_head = list_head->next();

    delete ptr;

    return e;
}
```



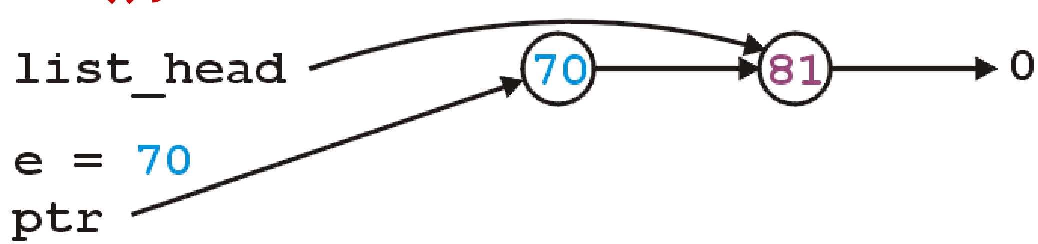
# Pop Front Definition

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();

    Node *ptr = list_head;

    list_head = list_head->next();

    delete ptr;

    return e;
}
```



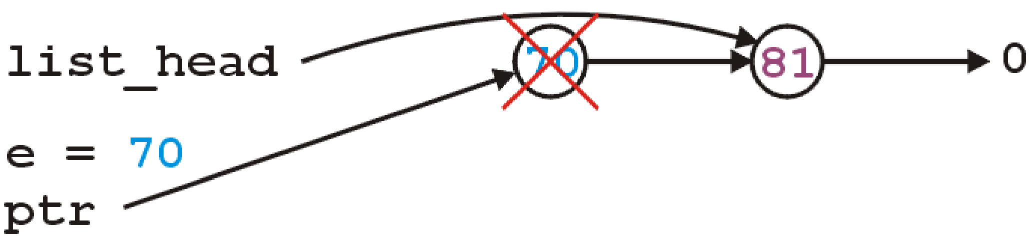
# Pop Front Definition

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();

    Node *ptr = list_head;

    list_head = list_head->next();

    delete ptr;

    return e;
}
```



# Pop Front Definition

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

     int e = front();

     Node *ptr = list_head;

     list_head = list_head->next();

     delete ptr;

     return e;
}
```



# Project 1: Linked List – Accessors and push_front

- Notice that the constructor is given to you
- Complete accessors in class Single_list.h (except `count`)

```
int size() const;
bool empty() const;
Object front() const;
Object back() const;
Single_node<Object> *head() const;
Single_node<Object> *tail() const;
```

- Complete the mutator push_front
- Test these functions using commands listed in Single_list_tester.h.

# Testing

- Test file with functions implemented so far, ie.

```
new
push_front 3
push_front 7
front 7
back 3
head
  retrieve 7
  next
  retrieve 3
  next0
exit
size 2
empty 0
Exit
```

- Try command “cout” – to list the content of your list
- Notice file int.in. If your code passes this test you have 50 % of the total mark.

# Stepping through Linked Lists

- The next step is to look at member functions which potentially require us to step through the entire list:

```
int count( int ) const;
int  erase( int );
```

- For count, we will return 1 if the object was found and 0 otherwise
- For erase, we return 1 if the object was found and 0 otherwise and we will delete the node.

# Stepping through Linked Lists

- The process of stepping through a linked list can be thought of as being analogous to a for loop:
  - We initialize a temporary pointer with the list head
  - We continue iterating until the pointer equals `0`
  - With each step, we set the pointer to point to the next object

# Stepping through Linked Lists

- Thus, we have:

```
for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    // Do something
    //  - use ptr->fn() to call member functions
    //  - use ptr->var to assign/access member variables
}
```

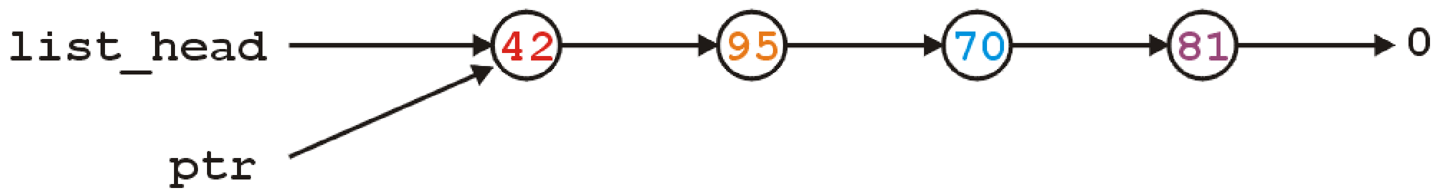
# Stepping through Linked Lists

- Analogously:

```
for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
   // do something
    // use ptr->fn() to call member functions
    // use ptr->var to assign/access member variables
}

for ( int i = 0; i < N; ++i ) {
    // do something
}
```

# Stepping through Linked Lists

- With the initialization and first iteration of the loop, we have:

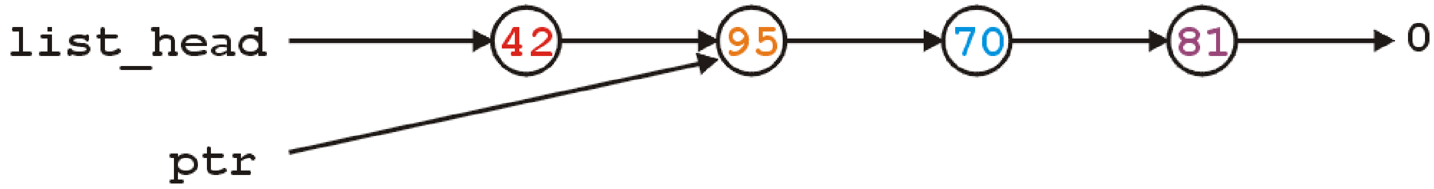



- **`ptr != 0`** and thus we evaluate the body of the loop and then set **`ptr`** to the next node: `ptr = ptr->next()`

```
for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    // Do something
}
```

# Stepping through Linked Lists

- **`ptr != 0`** and thus we evaluate the loop and increment the pointer

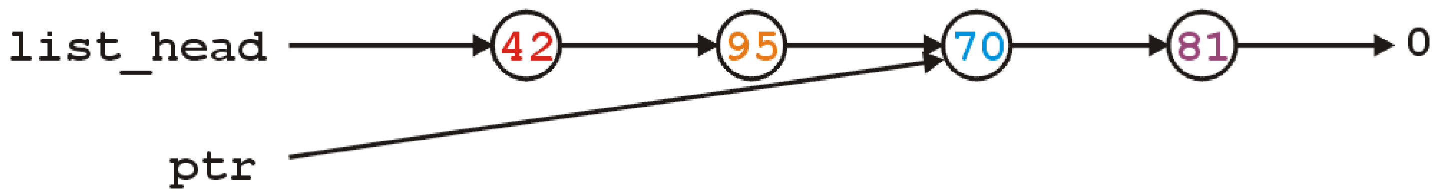



- In the loop, we can access 95 by using **`ptr->retrieve()`**

```
for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    // Do something
}
```

# Stepping through Linked Lists

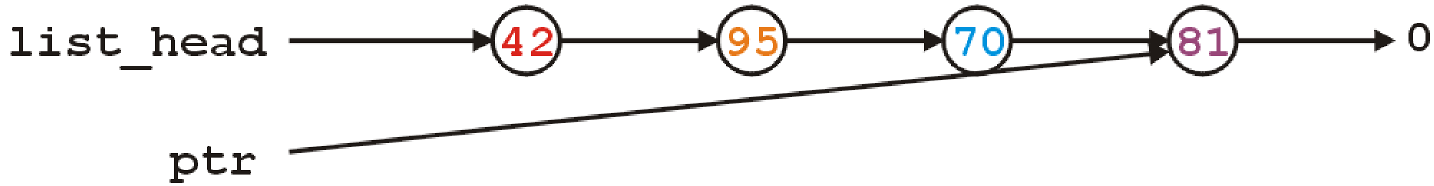
- **`ptr != 0`** and thus we evaluate the loop and increment the pointer




```
for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    // Do something
}
```

# Stepping through Linked Lists

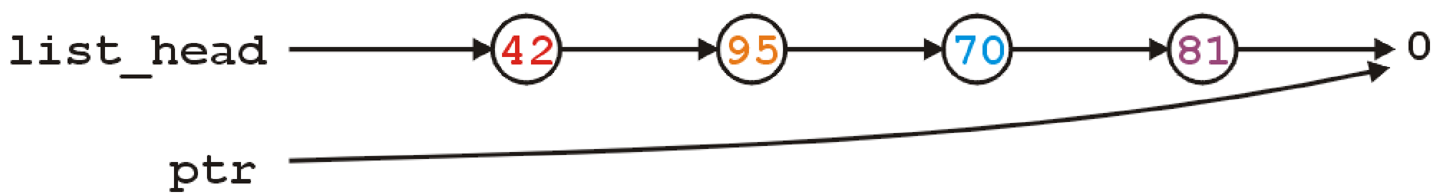
- `ptr != 0` and thus we evaluate the loop and increment the pointer




```
for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    // Do something
}
```

# Stepping through Linked Lists

- Here, we check and find **`ptr != 0`** is false, and thus we exit the loop




- Because the variable **`ptr`** was declared inside the loop, we can no longer access it

```
for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    // Do something
}
```

# Stepping through Linked Lists

- To implement `bool count(int) const;`, we check if the argument matches the retrieved element with each step
  - If we find the argument, we return `true`
  - If we finish the loop, this indicates we did not find the argument: return `false`

# Stepping through Linked Lists

- The implementation:

```
int List::count( int n ) const {
    for ( Node *ptr = head(); ptr != 0; ptr = ptr->next() ) {
        if ( ptr->retrieve() == n ) {
            return 1;
        }
    }

    return 0;
}
```

# Stepping through Linked Lists

- To erase an arbitrary element, i.e., to implement **`int erase( int )`**, we must update the previous node
- For example, given


if we delete **70**, we want to end up with

# Stepping through Linked Lists

- Notice that the **`erase`** function must modify the member variables of the node prior to the node being removed
- Thus, it must have access to the private instance variable **`next_node`**
- We could supply the member function

```
void set_next( Node * );
```

however, this would be globally accessible

# Stepping through Linked Lists

- In Java/C#, you could define the **`Node`** class to be an *inner* class
  - this is an elegant OO solution
- In C++, you explicitly break encapsulation by declaring the class List to be a *friend* of the class **`Node`**:

```
class Node {
Node * next() const;
    // ... declaration ...
    friend class List;
};
```

# Stepping through Linked Lists

- Now, inside erase (a member function of **`List`**), you can modify all the member variables of any instance of the **`Node`** class
  - Challenge for erase: you will need an additional temporary pointer when removing a node—once you update the linked list, you must delete the removed node

# The Destructor

- We dynamically allocated memory each time we added a new Node into this list
- Suppose we delete a list before it is empty
  - By default, this does not clean up any of those remaining nodes
- Thus, we need to define a destructor:

```
class List {
    private:
        Node *list_head;
    public:
        List();
        ~List();
        // ...etc...
};
```

# The Destructor

- The destructor has to delete any memory which was allocated with `new` but not yet deallocated
- This is straight-forward enough:

```
while ( !empty() ) {
    pop_front();
}
```

- The overhead of `pop_front` is negligible when compared to calling `delete`

# Project 1: count, erase and destructor

- Complete count
- **Test**
- Complete pop_front
- Complete erase, and destructor
- **Test**

# Assignment and Copy Constructor

- We now have the functionality of the linked list class
- However, there are other functional operations required for linked lists to interact with C++
- Recall two common C++ operations:
  - Assignment
  - Passing by value (the *copy* constructor)

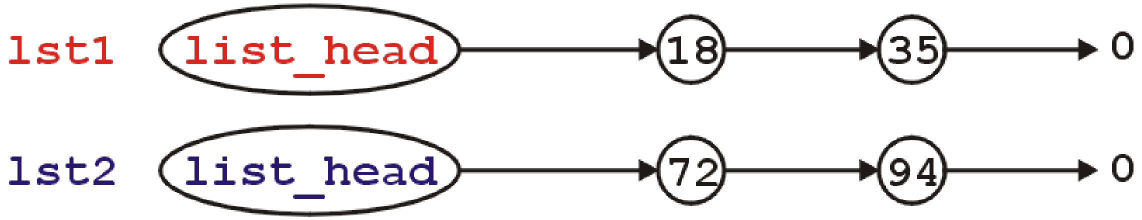
# Assignment and Copy Constructor

- Suppose you have two objects, be they linked lists or otherwise, for example:

```
List lst1;
List lst2;

lst1.push_front( 35 );
lst1.push_front( 18 );
lst2.push_front( 94 );
lst2.push_front( 72 );
```

# Assignment and Copy Constructor

- This is the current state:




- If we assign

  `lst2 = lst1;`

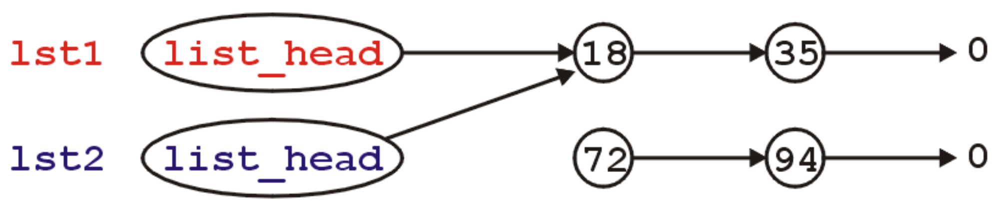
  by default, all that occurs is that the member variables from **`lst1`** are copied to the member variables of **`lst2`**

# Assignment and Copy Constructor

- Because the only member variable of this class is **`list_head`**, the value it is storing (the address of the first node) is copied over
- It is equivalent to writing:

```
lst2.list_head = lst1.list_head;
```

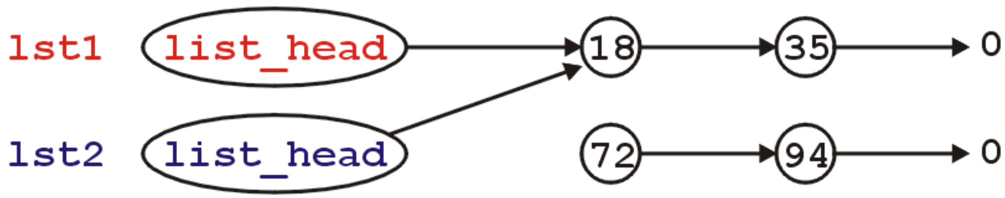
- Graphically:



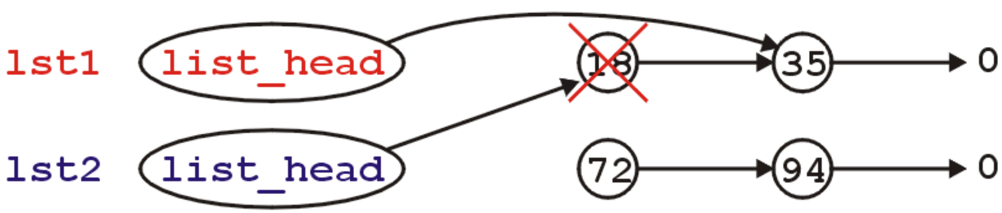
# Assignment and Copy Constructor

- What's wrong with this picture?
  - We no longer have links to either of the nodes storing 72 or 94
  - Also, suppose we call the member function

    `lst1.pop_front();`
  - This only affects the member variable from the object `lst1`
  - Thus, we go from




to



# Assignment and Copy Constructor

- Now, the second list **`lst2`** is pointing to memory which has been deallocated...
- What is the behavior is we call

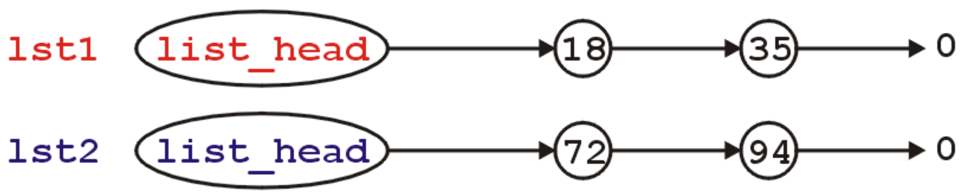
```
lst2.pop_front();
```

?

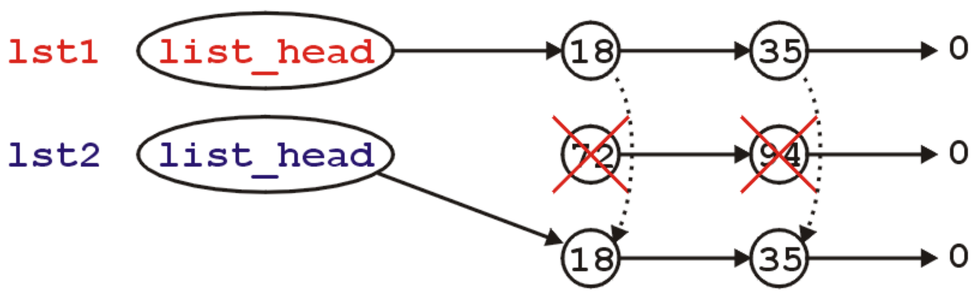
- The behaviour is undefined, however, soon this will probably lead to an access violation

# Assignment and Copy Constructor

- We need to properly implement assignment
- The logical solution would be to:
  - Delete all the nodes in `lst2`
  - Copy all of the nodes in `lst1` to `lst2`
  - Thus, we go from




to



# Assignment and Copy Constructor

- To define assignment, we must overload the function named **`operator=`**
  - This is a how you indicate to the compiler that you are overloading the **`=`** operator
- Its signature is:

```
List &operator=( List const & );
```

- To avoid copies, the argument and return value are passed by reference

```
List &operator=( const List &rhs );
```

- Because it is a reference, that means that you must refer use, for example, **`rhs.head()`** or **`rhs.push_front()`**

# Assignment and Copy Constructor

- Because we are overwriting the current object, we must:

```
List &operator=( List const &rhs ) {
    // clean up this object
    // make a complete copy of rhs

    return *this;
}
```

- Here, the name of the argument **`rhs`** refers to the right-hand side of the assignment

# Assignment and Copy Constructor

- To clean up the current linked list, we need to remove all nodes in this linked list:

```
List &operator=( List const &rhs ) {
    while ( !empty() ) {
        pop_front();
    }
    // Make a complete copy of rhs

    return *this;
}
```

# Assignment and Copy Constructor

- However, we have a problem: consider

```
List a;
a.push_front( 3 );
a = a;

List &operator= ( const List &rhs ) {
    while ( !empty() ) {
        pop_front();
    }
    // Both rhs and *this refer to the same
    // linked list which is now empty
    // Make a complete copy of rhs

    return *this;
}
```

# Assignment and Copy Constructor

- To avoid this problem, we must always make the following check:

```
List &operator= ( const List &rhs ) {
    if ( this == &rhs ) {
        return *this;          // No copy is necessary
    }
    while ( !empty() ) {
        pop_front();
    }
    // Make a complete copy of rhs

    return *this;
}
```

# Assignment and Copy Constructor

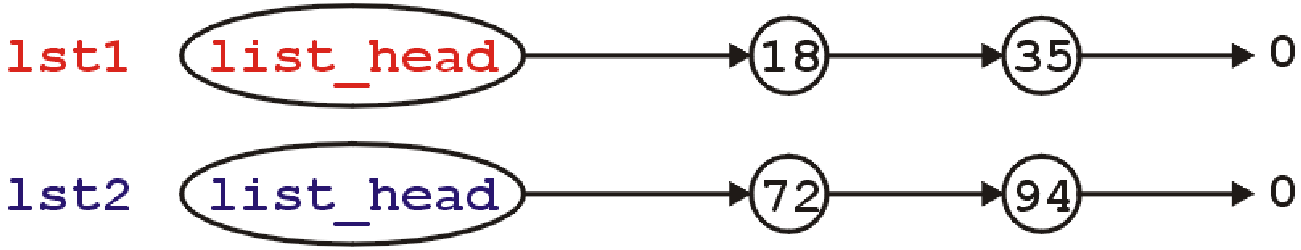
- To make a copy of the right hand side is a little more difficult...
- We must step through the second linked list and create a new node for each object in the **`rhs`** list

# Assignment and Copy Constructor

```
List &List::operator= ( const List &rhs ) {
    while ( !empty() ) {
        pop_front();
    }

    if ( rhs.empty() ) {
        return * this;
    }

    list_head = new Node( rhs.front() );  // copy the front element

    for ( Node * lhptr = head(), * rhptr = rhs.head();
        rhptr -> next() != 0;
        rhptr = rhptr -> next(), lhptr = lhptr -> next()
    ) {
        lhptr -> next_node = new Node( rhptr -> next() -> retrieve() );
    }

    return * this;
}
```

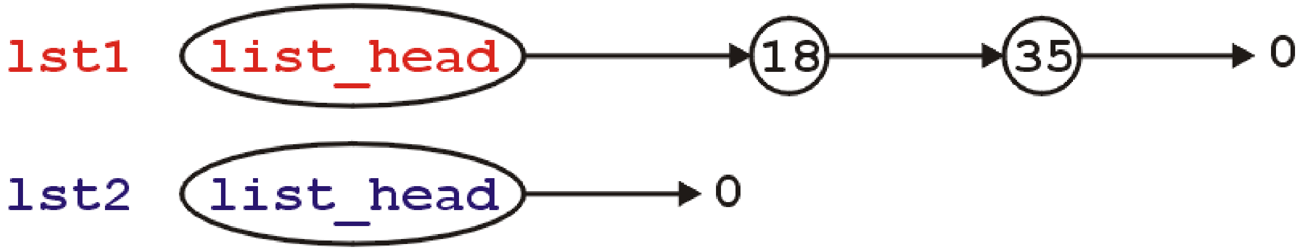
# Assignment and Copy Constructor

- Visually, we are doing the following:
- In assigning `lst2 = lst1;`



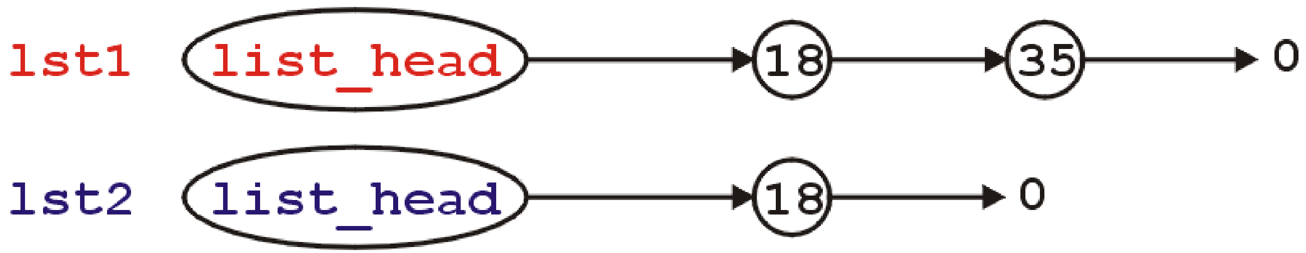
# Assignment and Copy Constructor

- Empty the second list:



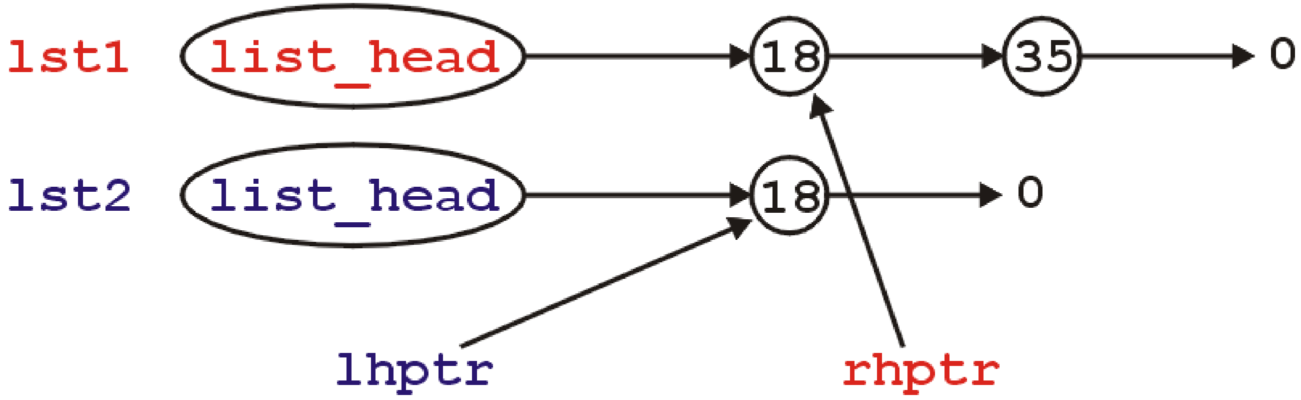
# Assignment and Copy Constructor

- If the 1st list is not empty, push the front element of the 1st list onto the 2nd list:



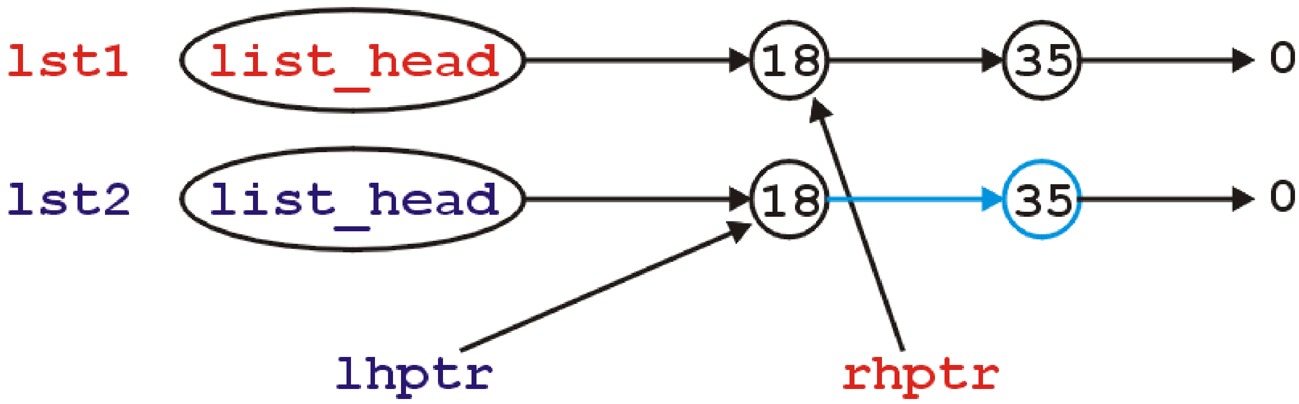
# Assignment and Copy Constructor

- Get two pointers pointing to the first nodes in each of the lists



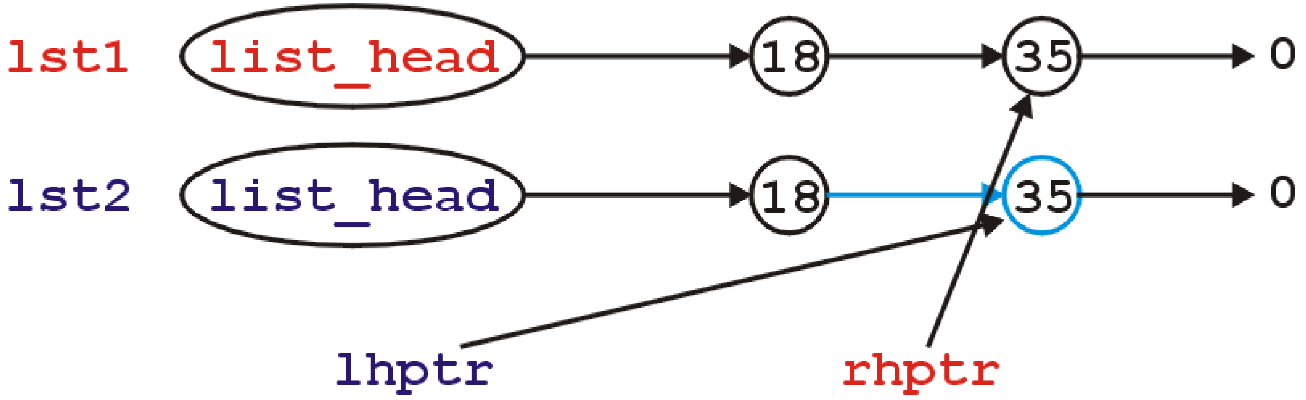
# Assignment and Copy Constructor

- While there is a node following the node pointed to by **`rhptr`**, add that node to the 2nd list:



# Assignment and Copy Constructor

- Then, increment each of the pointers and continue repeating this procedure until there are no more nodes to add



# Assignment and Copy Constructor

- Because in Project 1, you have a tail pointer, you can actually do this a lot more easily
- Like the member function, step through the first linked list, however, just call **`push_back( ptr -> retrieve() )`** with node in the list

# Assignment and Copy Constructor

- Why do we return `*this`?
- Suppose you have the assignment:

  `lst3 = lst2 = lst1;`
- The assignment operator is right associative: it first evaluates `lst2 = lst1;`
- After this, it must assign `lst3 = ?;`
- `?` is whatever is returned by the assignment operation of `lst2 = lst1;`

# Assignment and Copy Constructor

- We have discussed assignment
- We must now look at a similar problem:
  - Making a copy of an already-existing object (Ex1 and Ex2 below)
  - Passing or returning by value
- These are the same problem: an object is being constructed and initialized with the value of another.

Ex1.

```
List ls1;
Ls1.push_front(57);

List ls2( ls1 );
```

Ex2.

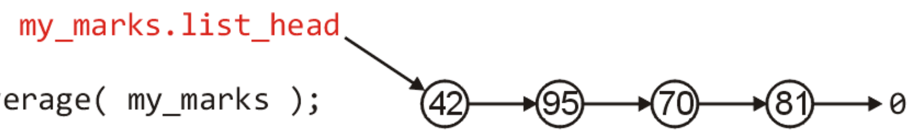
```
List ls1;
Ls1.push_front(57);

List ls2 = ls1;
```

# Assignment and Copy Constructor

- Consider this example where a linked list is passed to a function:
  – Four values are stored in the list

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```

```
cout << average( my_marks );
```




```
int average( List ls ) {


    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```

# Assignment and Copy Constructor

- By default, pass by value copies values
  - This includes all member variables are copied down including pointers which are just addresses

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```

my_marks.list_head

```
cout << average( my_marks );
```


Passed by value
- all member variables passed by value

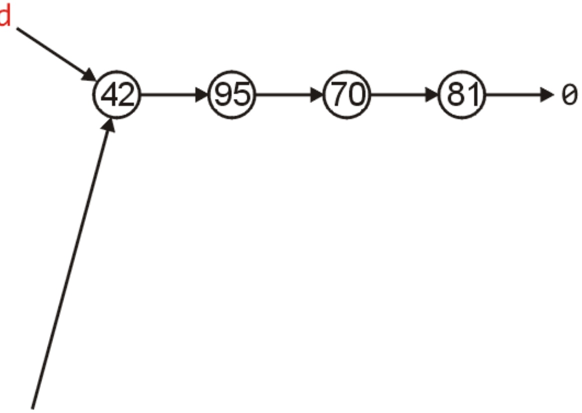
```
int average( List ls ) {


    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```

# Assignment and Copy Constructor

- Now we have two objects which store the same address in the member variable `list_head`

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```

my_marks.list_head




```
cout << average( my_marks );
```

```
int average( List ls ) {
```

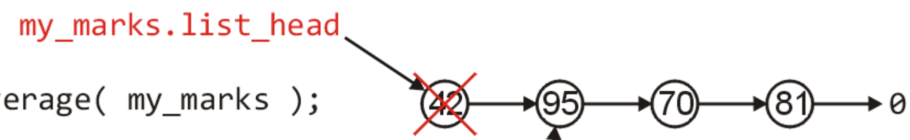
ls.list_head

```
    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```

# Assignment and Copy Constructor

- After one iteration of the loop, the pop_front function is called which deletes the first node

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```




```
cout << average( my_marks );
```

```
int average( List ls ) {

    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```

ls.list_head

# Assignment and Copy Constructor

- At the end of the function, all nodes have been deleted

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```

my_marks.list_head

```
cout << average( my_marks );
```

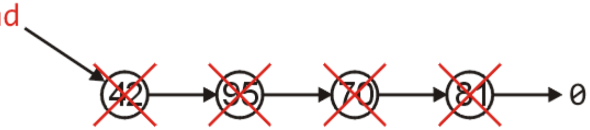

```
int average( List ls ) {

    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```

ls.list_head

# Assignment and Copy Constructor

- If, in the original function we now attempt to access a node, it no longer exists

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```




```
cout << average( my_marks );

cout << my_marks.front();
```

# Assignment and Copy Constructor

- We can, however, define a copy constructor
  - This function is called whenever a copy of the object is being made

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```

my_marks.list_head


```
cout << average( my_marks );
```

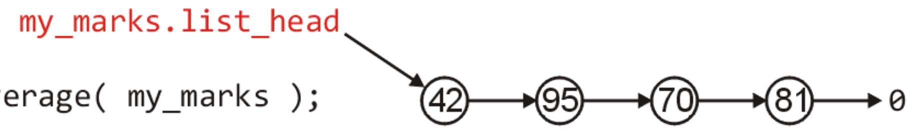
If a copy constructor exists, it is called
List::List( List const & );

```
int average( List ls ) {


    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```

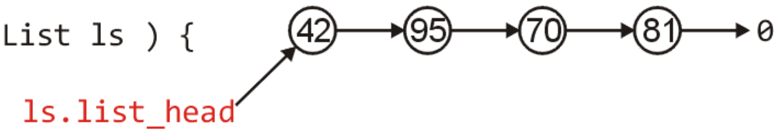
# Assignment and Copy Constructor

- In this case, the copy constructor should make a copy of all four nodes

```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```




```
cout << average( my_marks );
```




```
int average( List ls ) {

    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```

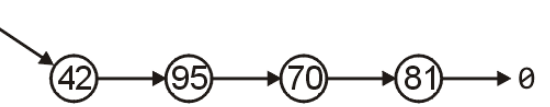
# Assignment and Copy Constructor

- Any changes made to this copy do not affect the original linked list

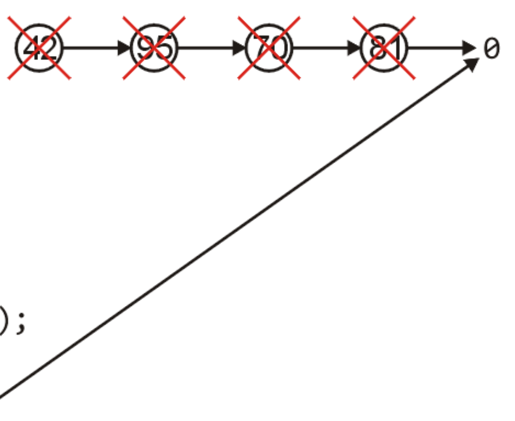
```
List my_marks;
my_marks.push_front( 81 );
my_marks.push_front( 70 );
my_marks.push_front( 95 );
my_marks.push_front( 42 );
```

my_marks.list_head

```
cout << average( my_marks );
```




```
int average( List ls ) {

    int n = 0, sum = 0;

    while ( !ls.empty() ) {
        sum += ls.pop_front();
        ++n;
    }

    return sum/n;
}
```




ls.list_head

# Assignment and Copy Constructor

- Thus, we require a *copy constructor*
- This is a method called when an object is being copied, either as a result of being passed-by-value to a function or being returned-by-value from a function
- Like a constructor, you should assume the variables are not initialized
- Unlike a constructor, you are not passed arguments to the constructor, rather, you are passed a reference to the object which should be copied

# Assignment and Copy Constructor

- The signature of a copy constructor has a constant reference to the same class as an argument:

```
List( const List & );
```

- The implementation of the copy constructor is almost identical to that of the **`operator =`**, except you are starting with a fresh object

# Assignment and Copy Constructor

- One simple implementation of a copy constructor is:

```
List::List( const List & lst ):list_head(0) {
    *this = lst;
}
```

- Here, all we do is:
  - initialize the array (set **list_head** to **0**), and
  - let **operator =** do all the work

# Summary

- First, things to remember:
  - In C++, a member function and a member variable cannot have the same name
  - Common mistakes:

```
ptr->retrieve;
list->head;
```

- Today, we looked at an example of a linked list implementation.

# Summary

- The Linked List example included
  - An overview of the list and node classes
  - Operations on linked lists
    - The default constructor
    - Getting started by coding simple functions
    - Implementing push front and pop front functions
    - Functions that require stepping through linked lists
    - The required behavior of the list destructor
    - The default and required behavior of operator=
    - The need for a copy constructor

# Summary

- The complete class is:

```
class List {
        private:
                Node * list_head;
        public:
                List();
                List( const List & );
                ~List();

                List & operator = ( const List & );

                bool empty() const;
                int front() const;
                Node * head() const;
                int count( int ) const;

                void push_front( int );
                int pop_front();
                int remove( int );
};
```

# Project 1: assignment operator and copy constructor

- Complete = operator
- Complete copy constructor
- **Test**

# Usage Notes

- These slides are made publicly available on the web for anyone to use
- If you choose to use them, or a part thereof, for a course at another institution, I ask only three things:
  - that you inform me that you are using the slides,
  - that you acknowledge my work, and
  - that you alert me of any mistakes which I made or changes which you make, and allow me the option of incorporating such changes (with an acknowledgment) in my set of slides

Sincerely,

Douglas Wilhelm Harder, MMath

**dwharder@alumni.uwaterloo.ca**