ECE 150 *Fundamentals of Programming*

Linked Lists

Douglas Wilhelm Harder, M.Math
Prof. Hiren Patel, Ph.D.
hdpatel@uwaterloo.ca dwharder@uwaterloo.ca

Outline

- In this lesson, we will:
  - Create a linked list class
  - Implement numerous member functions
  - Explain how to step through a linked list

A linked list

- The biggest issue with a linked list in the previous example is that the user has access to `p_list_head`
- Classes allow the user to prevent honest programmers from accessing its member variables

```
class Linked_list {
    private:
        Node *p_list_head_;
};
```

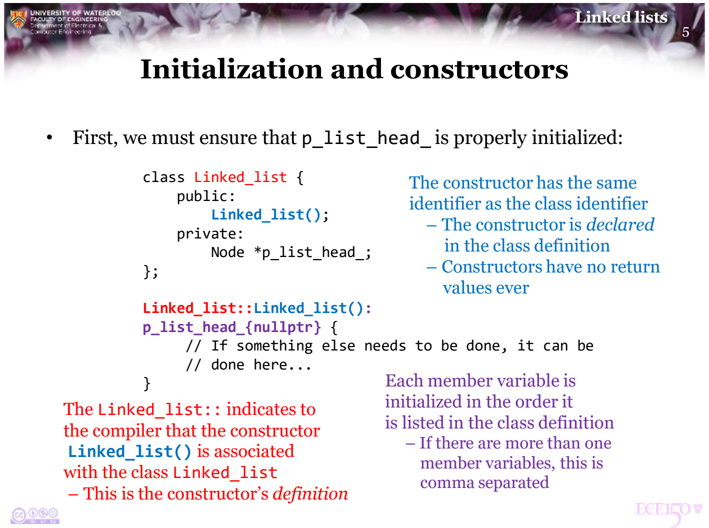
- The user can now create an instance of this linked list:

```
int main() {
    Linked_list my_list;

    // Do something with it...


    return 0;
}
```

Initialization and constructors

- First, we must ensure that `p_list_head_` is properly initialized:
  - It must be set to `nullptr`
- A class is associated with a *constructor*
  - This is a function that is automatically called when an instance is created

```
class Linked_list {
    public:
        Linked_list();
    private:
        Node *p_list_head_;
};

Linked_list::Linked_list():
p_list_head_{nullptr} {
     // If something else needs to be done, it can be
     // done here...
}
```



Initialization and constructors

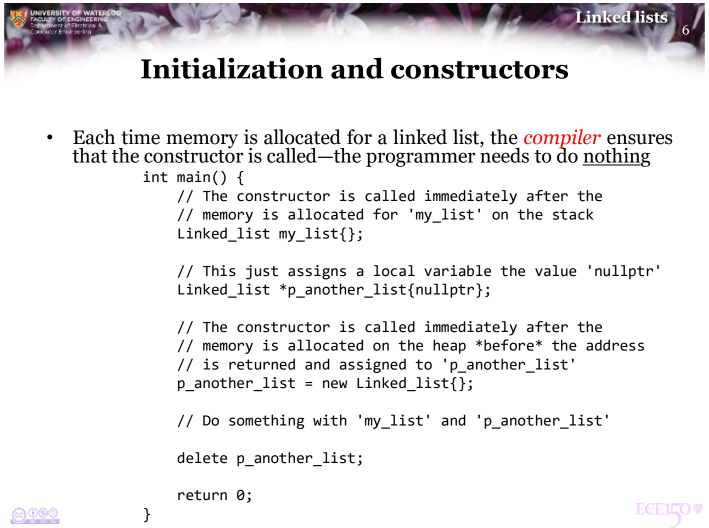
- First, we must ensure that `p_list_head_` is properly initialized:

```
class Linked_list {
    public:
        Linked_list();
    private:
        Node *p_list_head_;
};

Linked_list::Linked_list():
p_list_head_{nullptr} {
     // If something else needs to be done, it can be
     // done here...
}
```

The constructor has the same identifier as the class identifier
– The constructor is *declared* in the class definition
– Constructors have no return values ever

The `Linked_list::` indicates to the compiler that the constructor `Linked_list()` is associated with the class `Linked_list`
– This is the constructor's *definition*

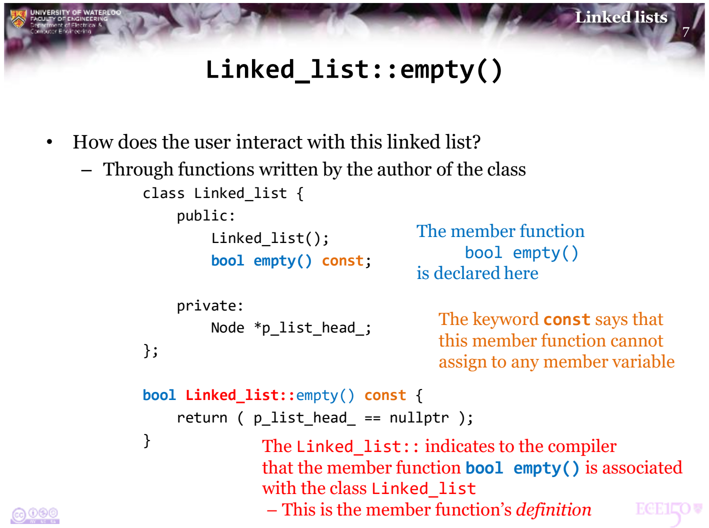
Each member variable is initialized in the order it is listed in the class definition
– If there are more than one member variables, this is comma separated



Initialization and constructors

- Each time memory is allocated for a linked list, the *compiler* ensures that the constructor is called—the programmer needs to do nothing

```
int main() {
    // The constructor is called immediately after the
    // memory is allocated for 'my_list' on the stack
    Linked_list my_list{};

    // This just assigns a local variable the value 'nullptr'
    Linked_list *p_another_list{nullptr};

    // The constructor is called immediately after the
    // memory is allocated on the heap *before* the address
    // is returned and assigned to 'p_another_list'
    p_another_list = new Linked_list{};

    // Do something with 'my_list' and 'p_another_list'

    delete p_another_list;

    return 0;
}
```



`Linked_list::empty()`

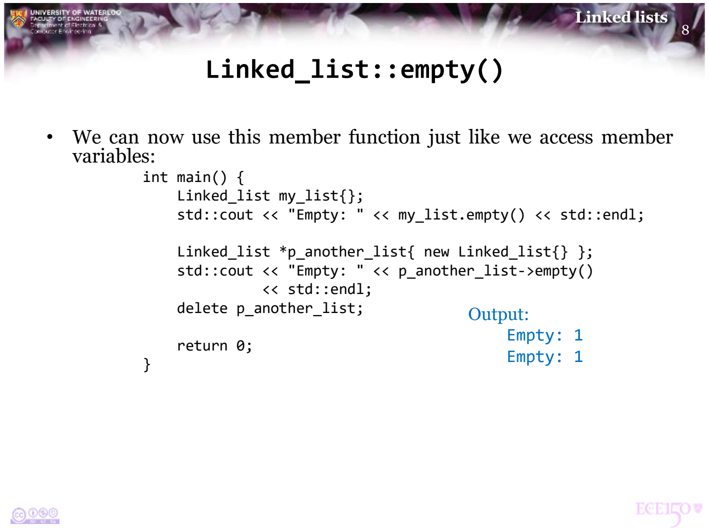
- How does the user interact with this linked list?
  - Through functions written by the author of the class

```
class Linked_list {
    public:
        Linked_list();
        bool empty() const;

    private:
        Node *p_list_head_;
};

bool Linked_list::empty() const {
    return ( p_list_head_ == nullptr );
}
```

The member function `bool empty()` is declared here

The keyword `const` says that this member function cannot assign to any member variable

The `Linked_list::` indicates to the compiler that the member function `bool empty()` is associated with the class `Linked_list`
– This is the member function's *definition*



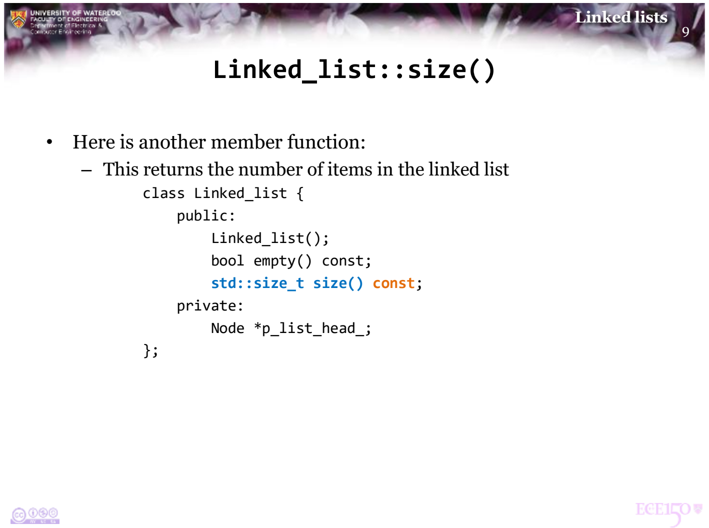
`Linked_list::empty()`

- We can now use this member function just like we access member variables:

```
int main() {
    Linked_list my_list{};
    std::cout << "Empty: " << my_list.empty() << std::endl;

    Linked_list *p_another_list{ new Linked_list{} };
    std::cout << "Empty: " << p_another_list->empty()
              << std::endl;
    delete p_another_list;

    return 0;
}
```

Output:

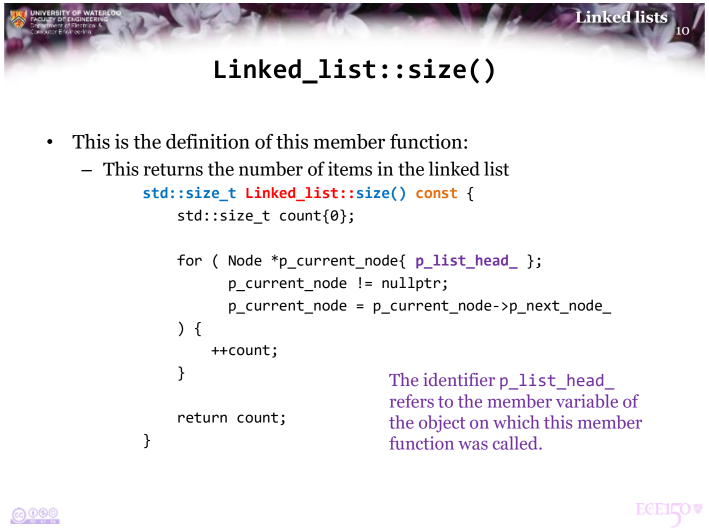
```
Empty: 1
Empty: 1
```



Linked_list::size()

- Here is another member function:
  - This returns the number of items in the linked list

```
class Linked_list {
    public:
        Linked_list();
        bool empty() const;
        std::size_t size() const;
    private:
        Node *p_list_head_;
};
```

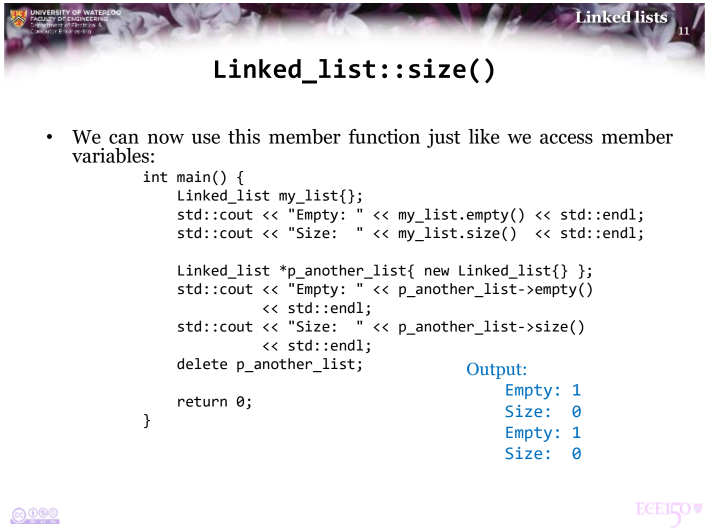


Linked_list::size()

- This is the definition of this member function:
  - This returns the number of items in the linked list

```
std::size_t Linked_list::size() const {
    std::size_t count{0};

    for ( Node *p_current_node{ p_list_head_ };
          p_current_node != nullptr;
          p_current_node = p_current_node->p_next_node_
    ) {
        ++count;
    }

    return count;
}
```

The identifier p_list_head_ refers to the member variable of the object on which this member function was called.



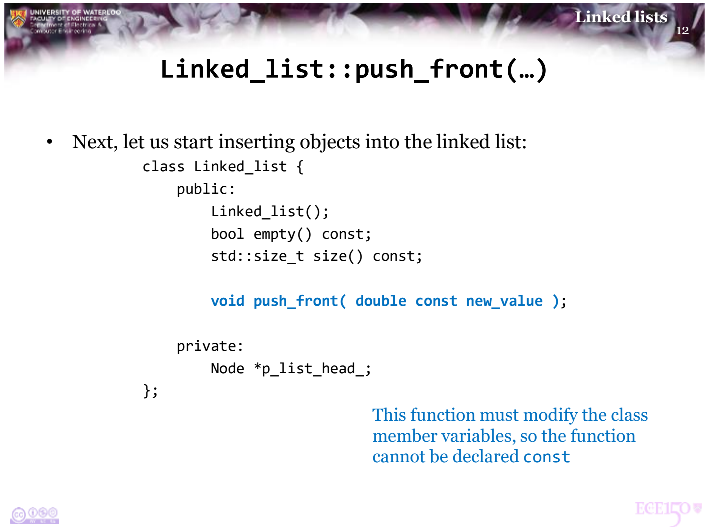
Linked_list::size()

- We can now use this member function just like we access member variables:

```
int main() {
    Linked_list my_list{};
    std::cout << "Empty: " << my_list.empty() << std::endl;
    std::cout << "Size:  " << my_list.size()  << std::endl;

    Linked_list *p_another_list{ new Linked_list{} };
    std::cout << "Empty: " << p_another_list->empty()
              << std::endl;
    std::cout << "Size:  " << p_another_list->size()
              << std::endl;
    delete p_another_list;

    return 0;
}
```

Output:

```
Empty: 1
Size:  0
Empty: 1
Size:  0
```

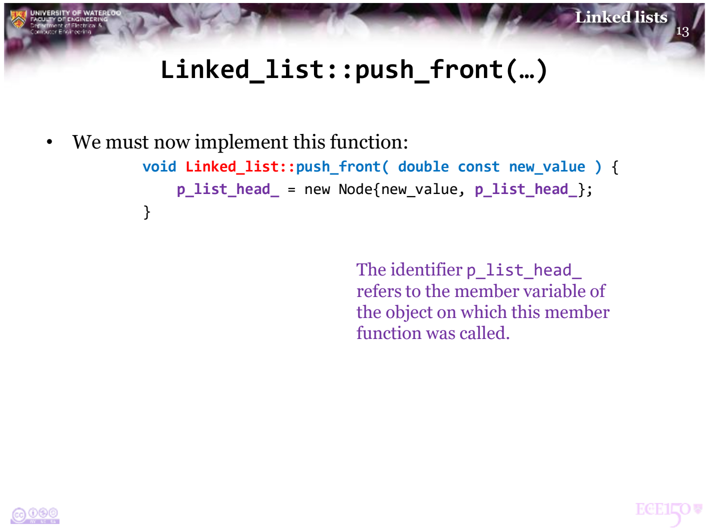


Linked_list::push_front(…)

- Next, let us start inserting objects into the linked list:

```
class Linked_list {
    public:
        Linked_list();
        bool empty() const;
        std::size_t size() const;

        void push_front( double const new_value );

    private:
        Node *p_list_head_;
};
```

This function must modify the class member variables, so the function cannot be declared const



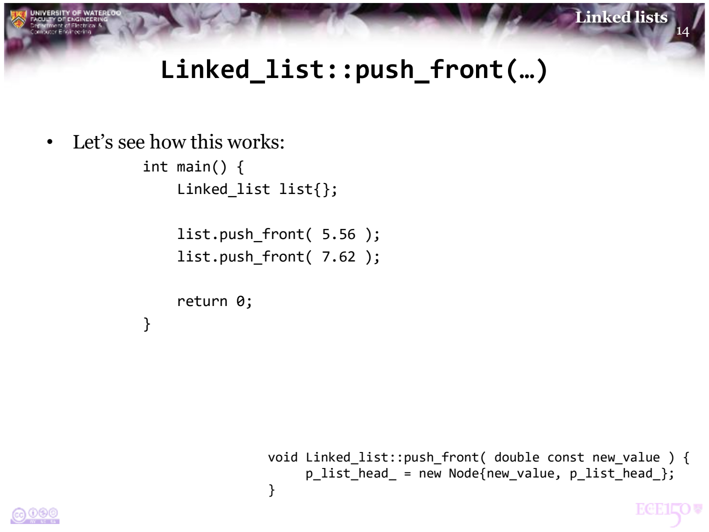
Linked lists 13

Linked_list::push_front(…)

- We must now implement this function:

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```

The identifier `p_list_head_` refers to the member variable of the object on which this member function was called.



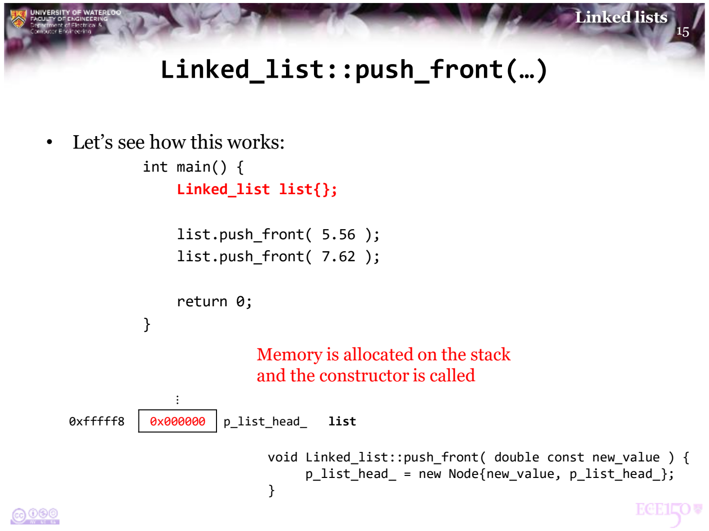
Linked lists 14

Linked_list::push_front(…)

- Let's see how this works:

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



Linked lists 15

Linked_list::push_front(…)

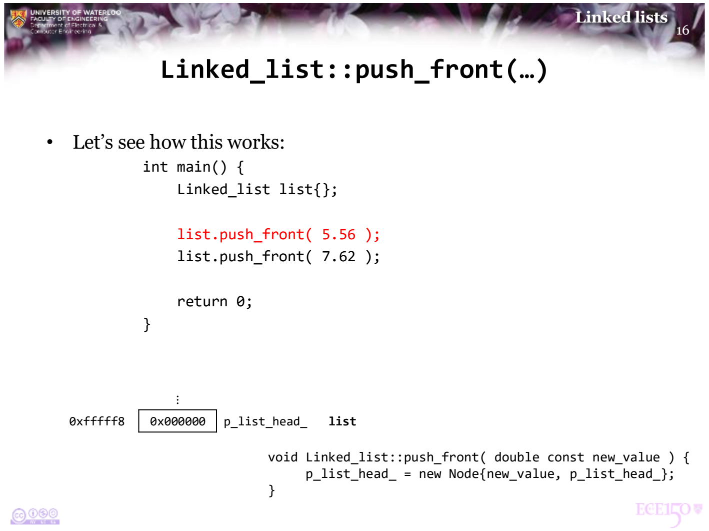
- Let's see how this works:

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

Memory is allocated on the stack and the constructor is called

```
             ⋮
0xffffff8 │0x000000│ p_list_head_   list
```

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



Linked lists 16

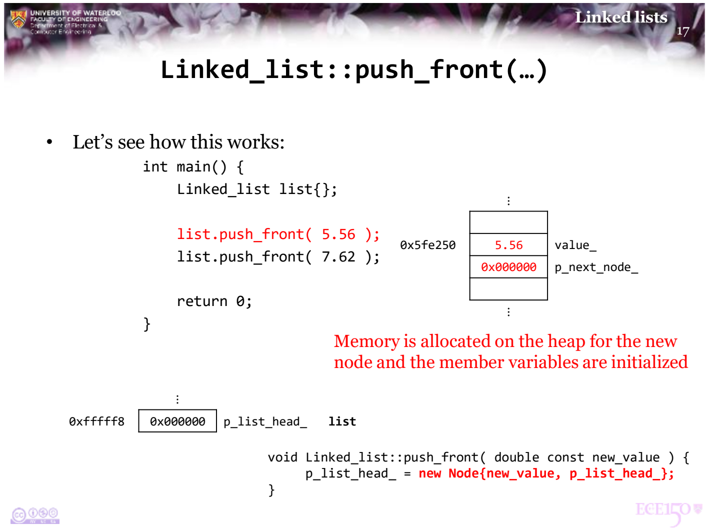
Linked_list::push_front(…)

- Let's see how this works:

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

```
             ⋮
0xffffff8 │0x000000│ p_list_head_   list
```

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



Linked_list::push_front(…)

- Let's see how this works:

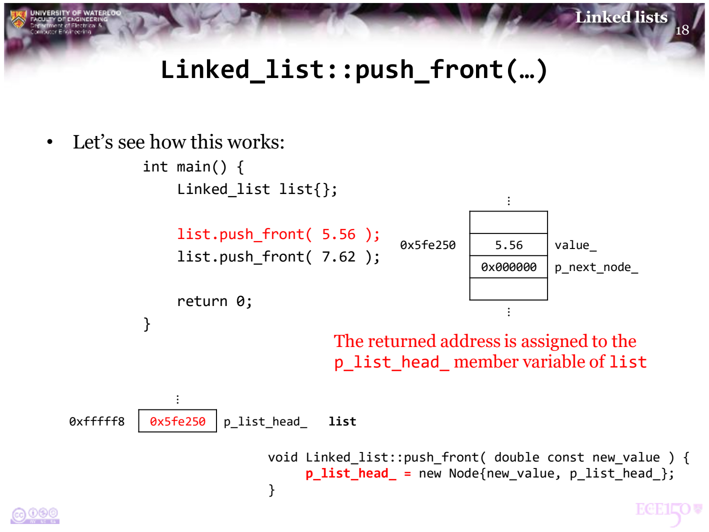
```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

0x5fe250 | 5.56 | value_
| 0x000000 | p_next_node_

Memory is allocated on the heap for the new node and the member variables are initialized

0xffffff8 | 0x000000 | p_list_head_ **list**

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



Linked_list::push_front(…)

- Let's see how this works:

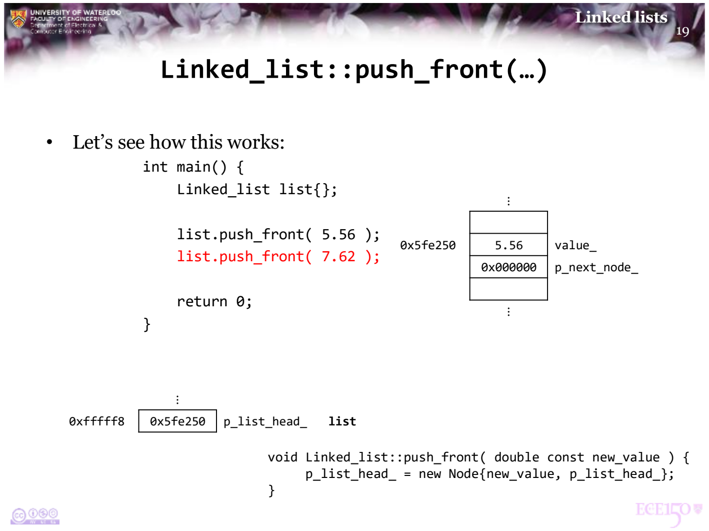
```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

0x5fe250 | 5.56 | value_
| 0x000000 | p_next_node_

The returned address is assigned to the `p_list_head_` member variable of `list`

0xffffff8 | 0x5fe250 | p_list_head_ **list**

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



Linked_list::push_front(…)

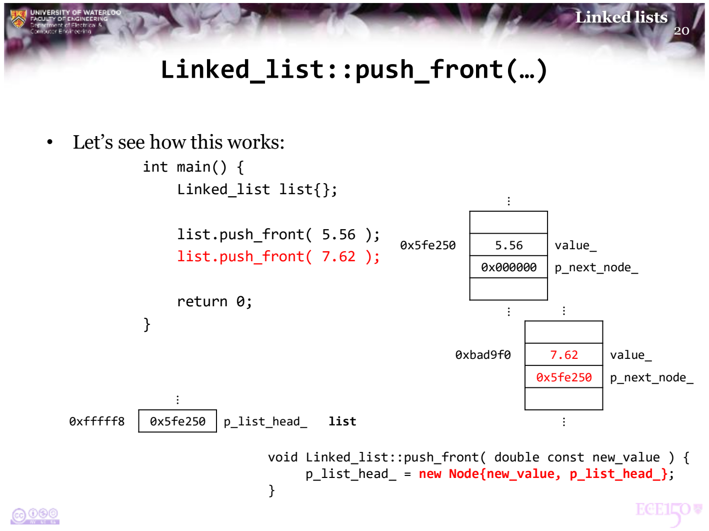
- Let's see how this works:

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

0x5fe250 | 5.56 | value_
| 0x000000 | p_next_node_

0xffffff8 | 0x5fe250 | p_list_head_ **list**

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



Linked_list::push_front(…)

- Let's see how this works:

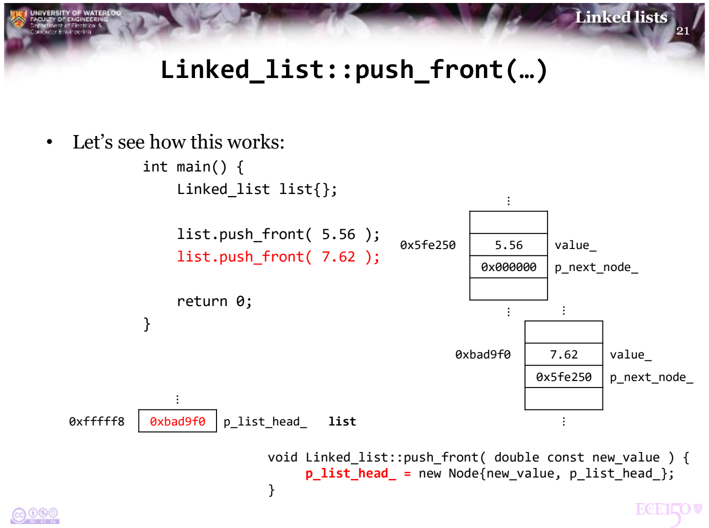
```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

0x5fe250 | 5.56 | value_
| 0x000000 | p_next_node_

0xbad9f0 | 7.62 | value_
| 0x5fe250 | p_next_node_

0xffffff8 | 0x5fe250 | p_list_head_ **list**

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



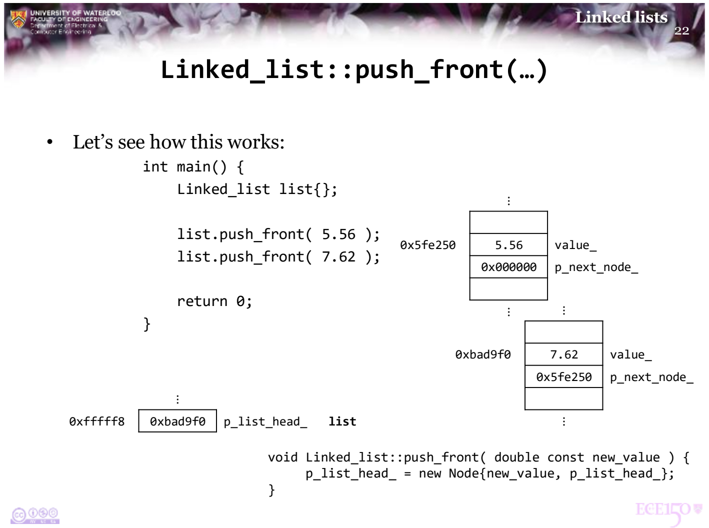
Linked_list::push_front(…)

- Let's see how this works:

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

```
                         ⋮
              0x5fe250 | 5.56     | value_
                       | 0x000000 | p_next_node_
                         ⋮          ⋮
                       0xbad9f0 | 7.62     | value_
                                | 0x5fe250 | p_next_node_
         ⋮
0xfffff8 | 0xbad9f0 | p_list_head_   list       ⋮
```

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```



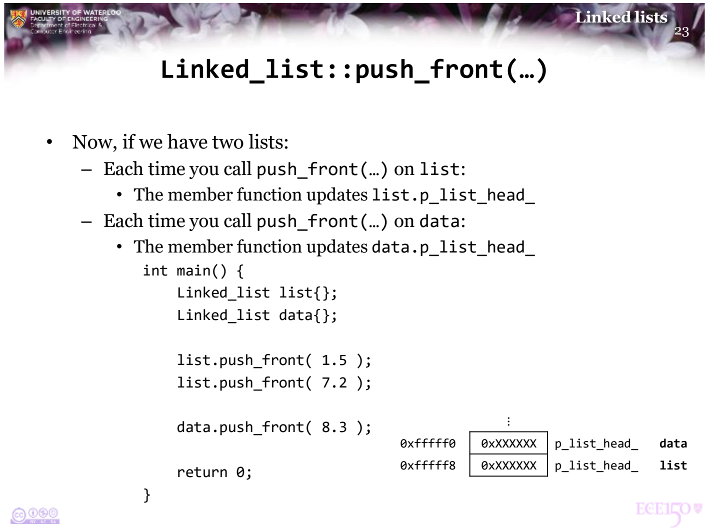
Linked_list::push_front(…)

- Let's see how this works:

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    return 0;
}
```

```
                         ⋮
              0x5fe250 | 5.56     | value_
                       | 0x000000 | p_next_node_
                         ⋮          ⋮
                       0xbad9f0 | 7.62     | value_
                                | 0x5fe250 | p_next_node_
         ⋮
0xfffff8 | 0xbad9f0 | p_list_head_   list       ⋮
```

```
void Linked_list::push_front( double const new_value ) {
    p_list_head_ = new Node{new_value, p_list_head_};
}
```

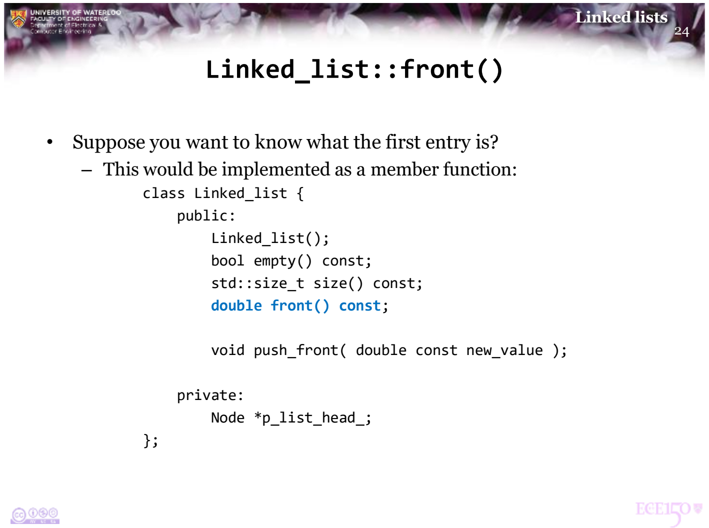


Linked_list::push_front(…)

- Now, if we have two lists:
  - Each time you call push_front(…) on list:
    - The member function updates list.p_list_head_
  - Each time you call push_front(…) on data:
    - The member function updates data.p_list_head_

```
int main() {
    Linked_list list{};
    Linked_list data{};

    list.push_front( 1.5 );
    list.push_front( 7.2 );

    data.push_front( 8.3 );

    return 0;
}
```

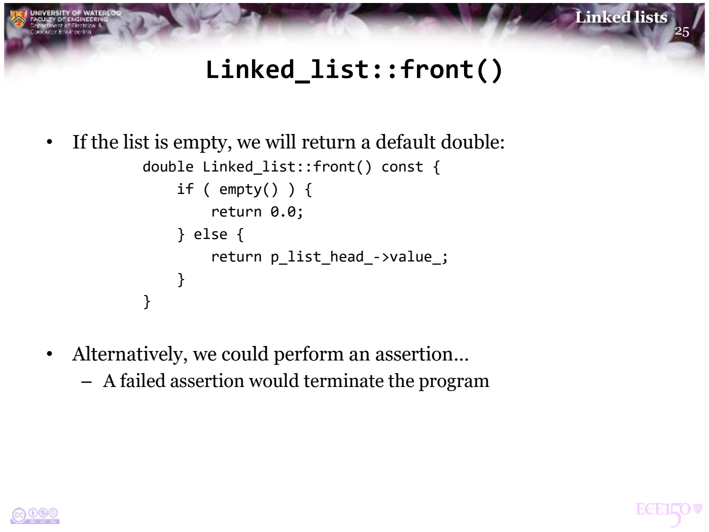
```
              ⋮
0xfffff0 | 0xXXXXXX | p_list_head_   data
0xfffff8 | 0xXXXXXX | p_list_head_   list
```



Linked_list::front()

- Suppose you want to know what the first entry is?
  - This would be implemented as a member function:

```
class Linked_list {
    public:
        Linked_list();
        bool empty() const;
        std::size_t size() const;
        double front() const;

        void push_front( double const new_value );

    private:
        Node *p_list_head_;
};
```

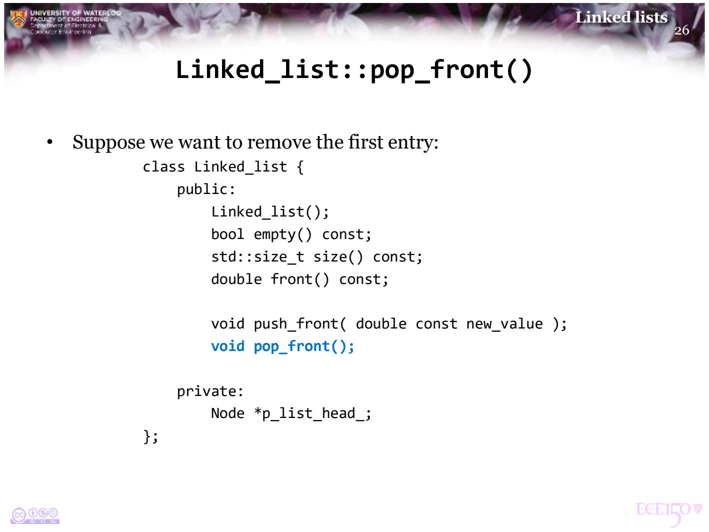


Linked_list::front()

- If the list is empty, we will return a default double:

```
double Linked_list::front() const {
    if ( empty() ) {
        return 0.0;
    } else {
        return p_list_head_->value_;
    }
}
```

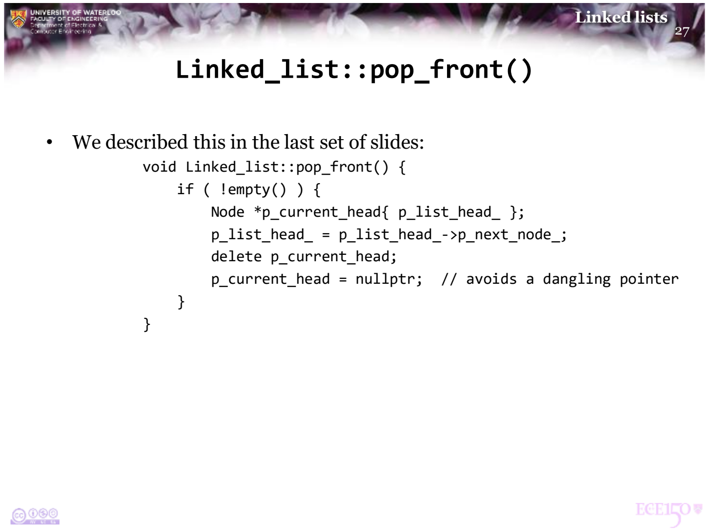
- Alternatively, we could perform an assertion…
  - A failed assertion would terminate the program



Linked_list::pop_front()

- Suppose we want to remove the first entry:

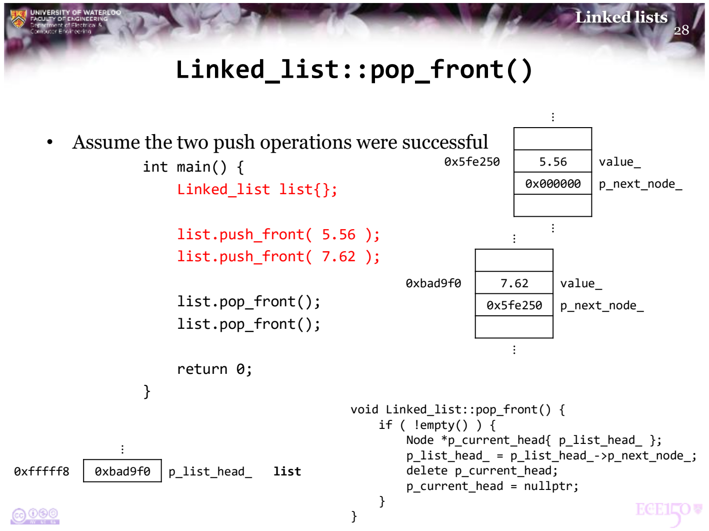
```
class Linked_list {
    public:
        Linked_list();
        bool empty() const;
        std::size_t size() const;
        double front() const;

        void push_front( double const new_value );
        void pop_front();

    private:
        Node *p_list_head_;
};
```



Linked_list::pop_front()

- We described this in the last set of slides:

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;  // avoids a dangling pointer
    }
}
```



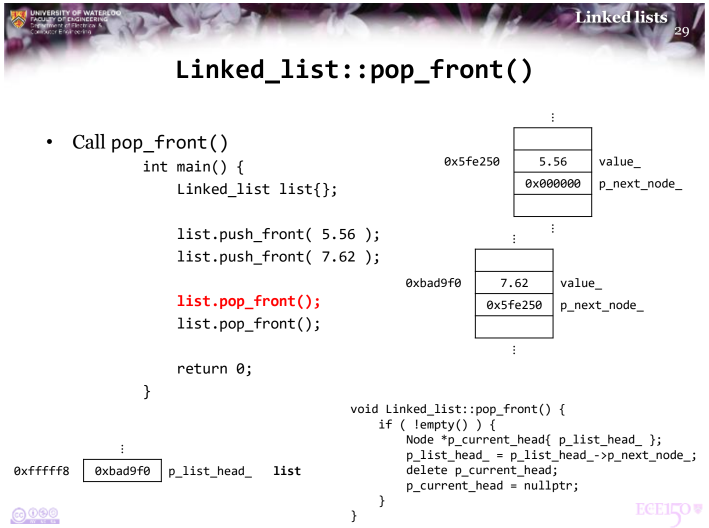
Linked_list::pop_front()

- Assume the two push operations were successful

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250
5.56
value_
0x000000
p_next_node_
0xbad9f0
7.62
value_
0x5fe250
p_next_node_
0xffffff8
0xbad9f0
p_list_head_
list

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 29

Linked_list::pop_front()

- Call pop_front()

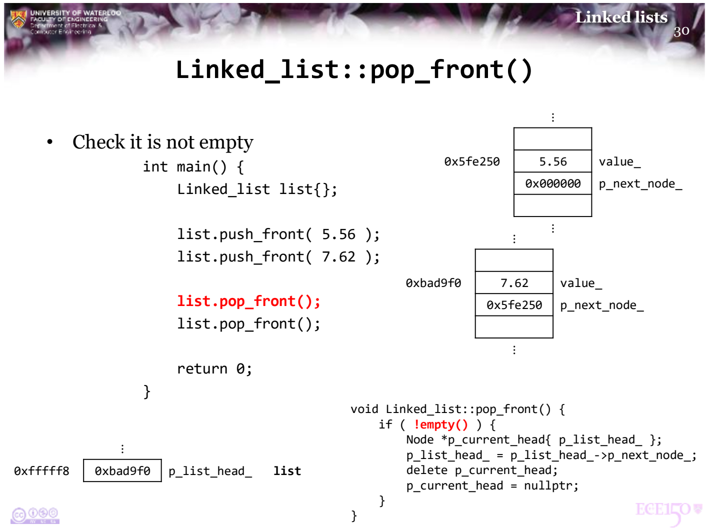
```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250 | 5.56 | value_
0x000000 | p_next_node_

0xbad9f0 | 7.62 | value_
0x5fe250 | p_next_node_

0xfffff8 | 0xbad9f0 | p_list_head_ **list**

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 30

Linked_list::pop_front()

- Check it is not empty

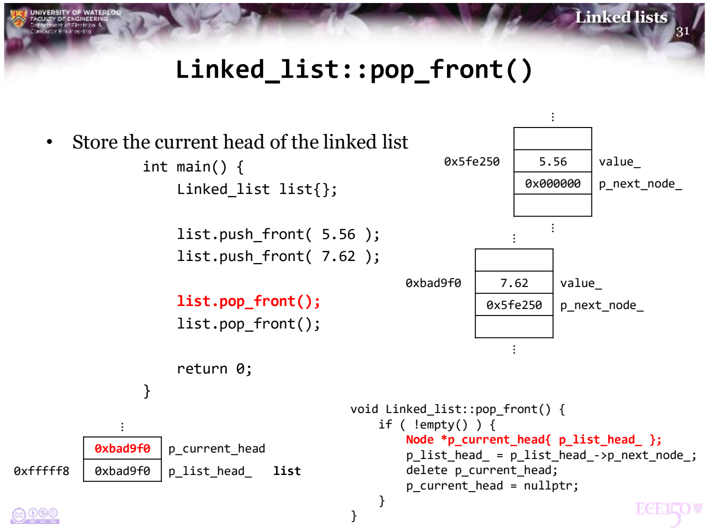
```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250 | 5.56 | value_
0x000000 | p_next_node_

0xbad9f0 | 7.62 | value_
0x5fe250 | p_next_node_

0xfffff8 | 0xbad9f0 | p_list_head_ **list**

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 31

Linked_list::pop_front()

- Store the current head of the linked list

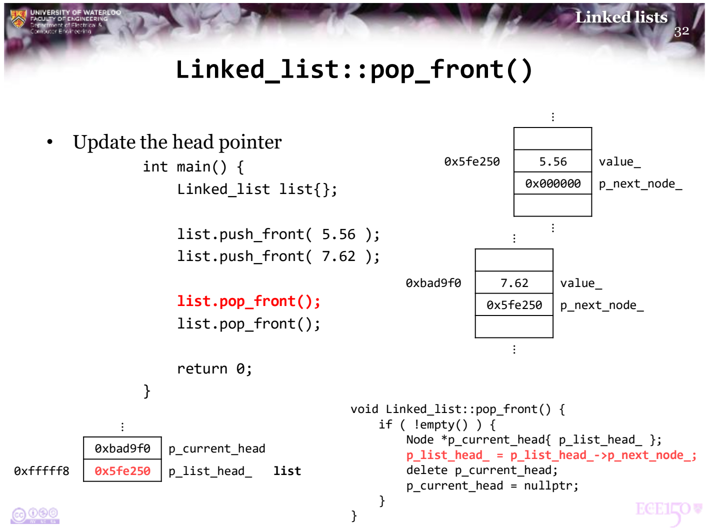
```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250 | 5.56 | value_
0x000000 | p_next_node_

0xbad9f0 | 7.62 | value_
0x5fe250 | p_next_node_

0xbad9f0 | p_current_head
0xfffff8 | 0xbad9f0 | p_list_head_ **list**

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 32

Linked_list::pop_front()

- Update the head pointer

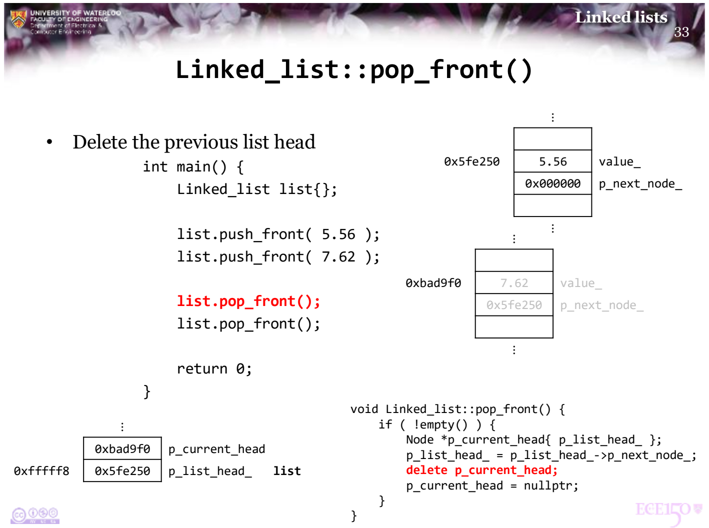
```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250 | 5.56 | value_
0x000000 | p_next_node_

0xbad9f0 | 7.62 | value_
0x5fe250 | p_next_node_

0xbad9f0 | p_current_head
0xfffff8 | 0x5fe250 | p_list_head_ **list**

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 33

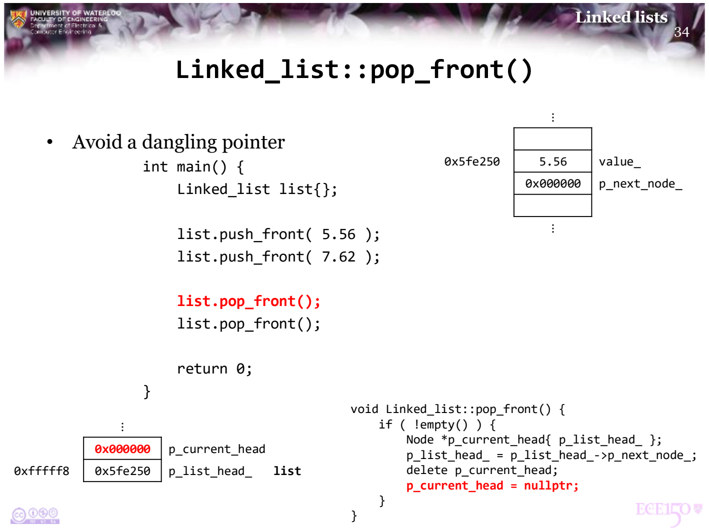
Linked_list::pop_front()

- Delete the previous list head

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250
5.56
value_
0x000000
p_next_node_
0xbad9f0
7.62
value_
0x5fe250
p_next_node_
0xbad9f0
p_current_head
0xffffff8
0x5fe250
p_list_head_
list

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 34

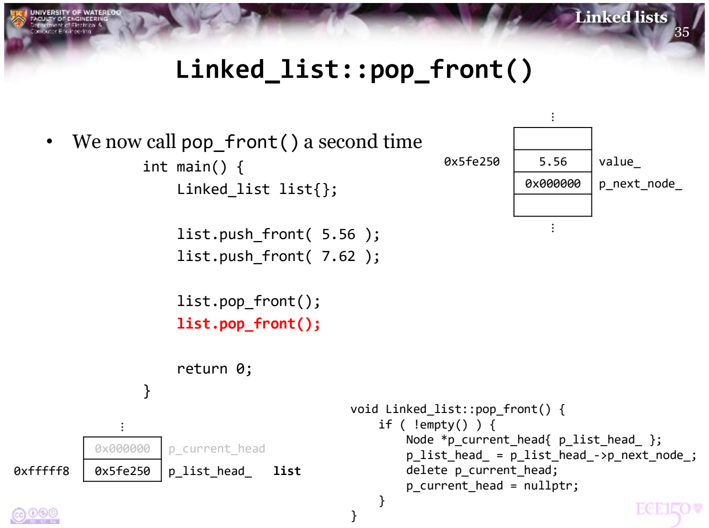
Linked_list::pop_front()

- Avoid a dangling pointer

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250
5.56
value_
0x000000
p_next_node_
0x000000
p_current_head
0xffffff8
0x5fe250
p_list_head_
list

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 35

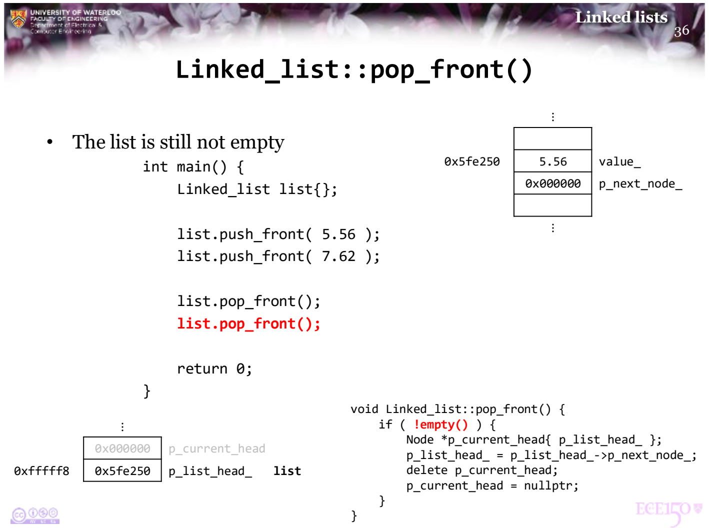
Linked_list::pop_front()

- We now call pop_front() a second time

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250
5.56
value_
0x000000
p_next_node_
0x000000
p_current_head
0xffffff8
0x5fe250
p_list_head_
list

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked lists 36

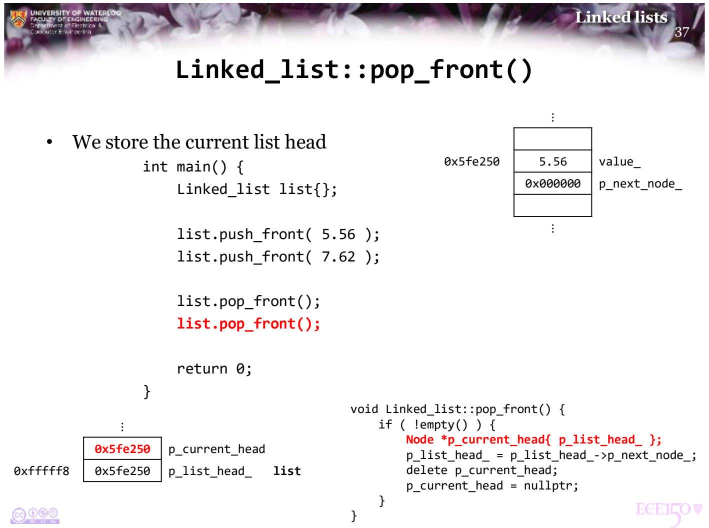
Linked_list::pop_front()

- The list is still not empty

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

0x5fe250
5.56
value_
0x000000
p_next_node_
0x000000
p_current_head
0xffffff8
0x5fe250
p_list_head_
list

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked_list::pop_front()

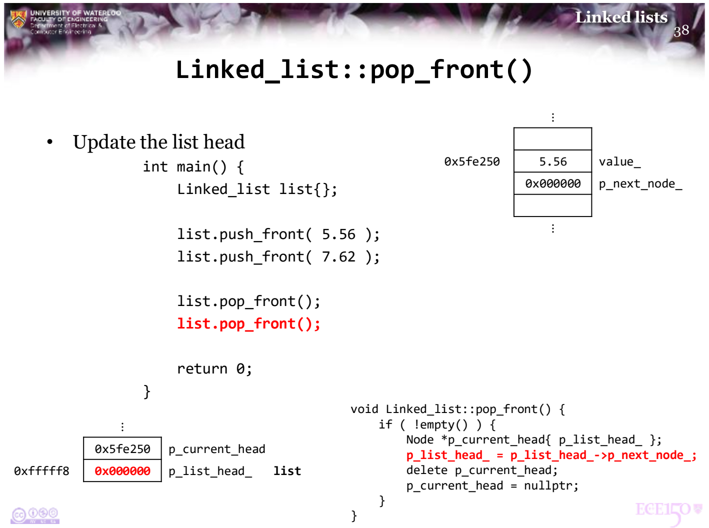
- We store the current list head

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

```
              ⋮
0x5fe250 |   5.56   | value_
         | 0x000000 | p_next_node_
              ⋮
```

```
              ⋮
         | 0x5fe250 | p_current_head
0xfffff8 | 0x5fe250 | p_list_head_   list
```

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked_list::pop_front()

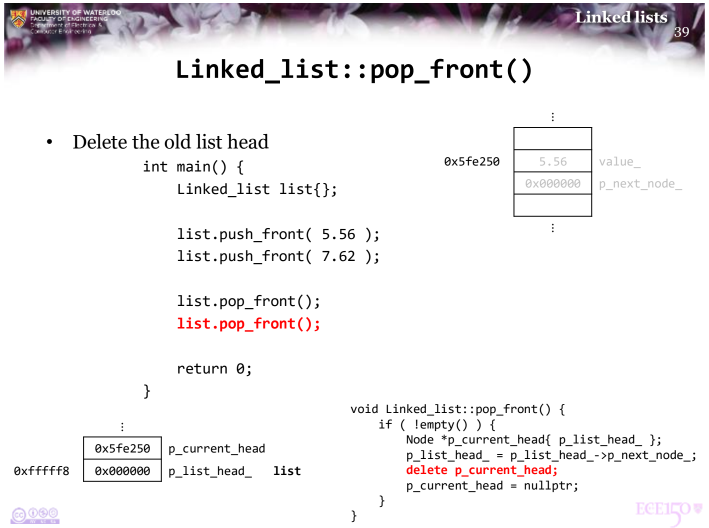
- Update the list head

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

```
              ⋮
0x5fe250 |   5.56   | value_
         | 0x000000 | p_next_node_
              ⋮
```

```
              ⋮
         | 0x5fe250 | p_current_head
0xfffff8 | 0x000000 | p_list_head_   list
```

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked_list::pop_front()

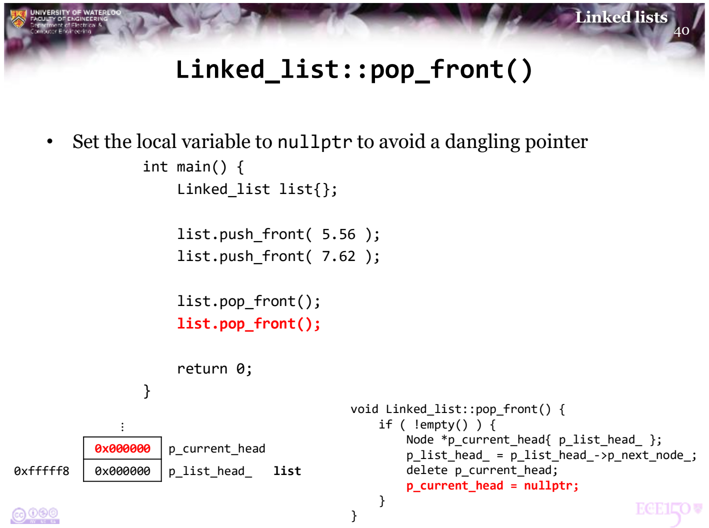
- Delete the old list head

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

```
              ⋮
0x5fe250 |   5.56   | value_
         | 0x000000 | p_next_node_
              ⋮
```

```
              ⋮
         | 0x5fe250 | p_current_head
0xfffff8 | 0x000000 | p_list_head_   list
```

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



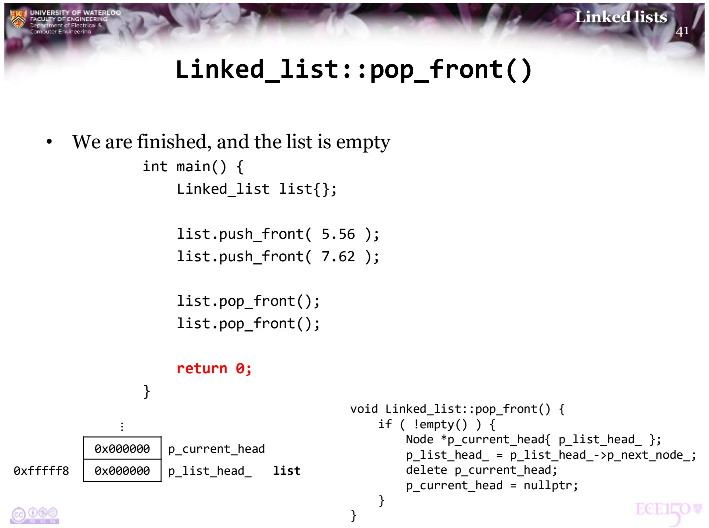
Linked_list::pop_front()

- Set the local variable to `nullptr` to avoid a dangling pointer

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

```
              ⋮
         | 0x000000 | p_current_head
0xfffff8 | 0x000000 | p_list_head_   list
```

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



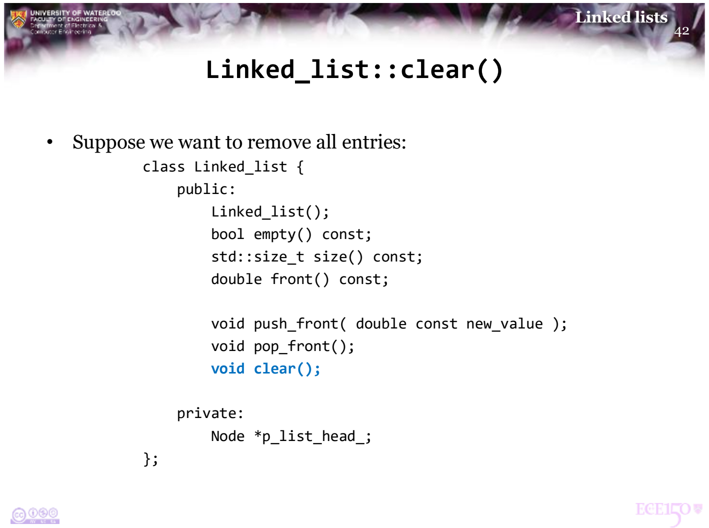
Linked_list::pop_front()

- We are finished, and the list is empty

```
int main() {
    Linked_list list{};

    list.push_front( 5.56 );
    list.push_front( 7.62 );

    list.pop_front();
    list.pop_front();

    return 0;
}
```

```
                  ⋮
          0x000000   p_current_head
0xfffff8  0x000000   p_list_head_    list
```

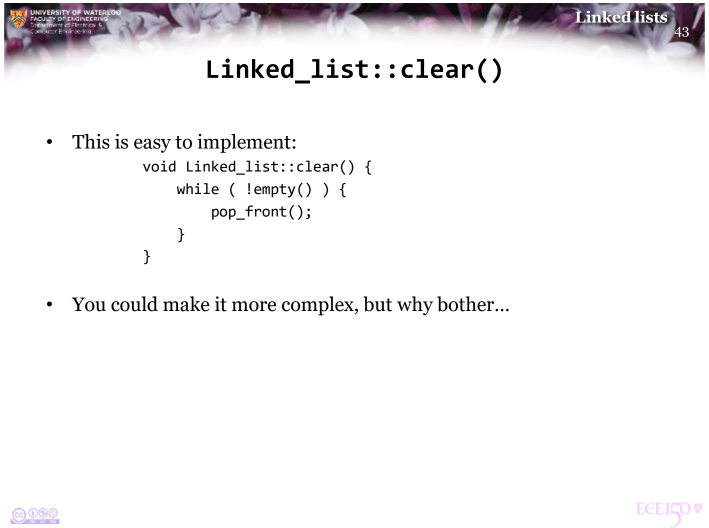
```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```



Linked_list::clear()

- Suppose we want to remove all entries:

```
class Linked_list {
    public:
        Linked_list();
        bool empty() const;
        std::size_t size() const;
        double front() const;

        void push_front( double const new_value );
        void pop_front();
        void clear();

    private:
        Node *p_list_head_;
};
```

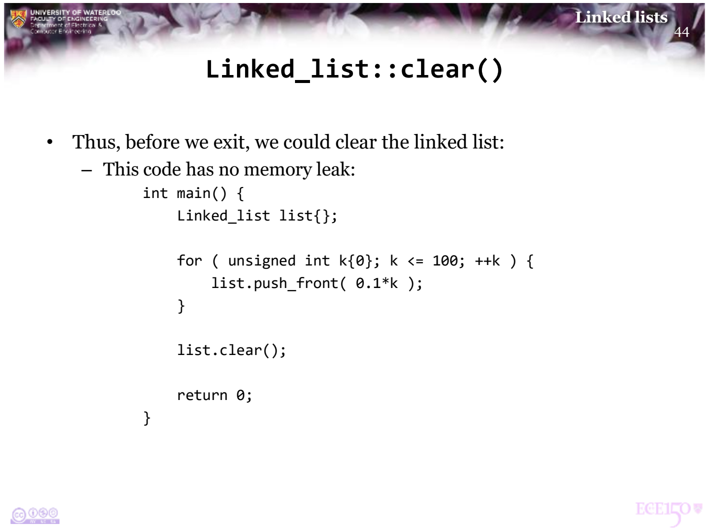


Linked_list::clear()

- This is easy to implement:

```
void Linked_list::clear() {
    while ( !empty() ) {
        pop_front();
    }
}
```

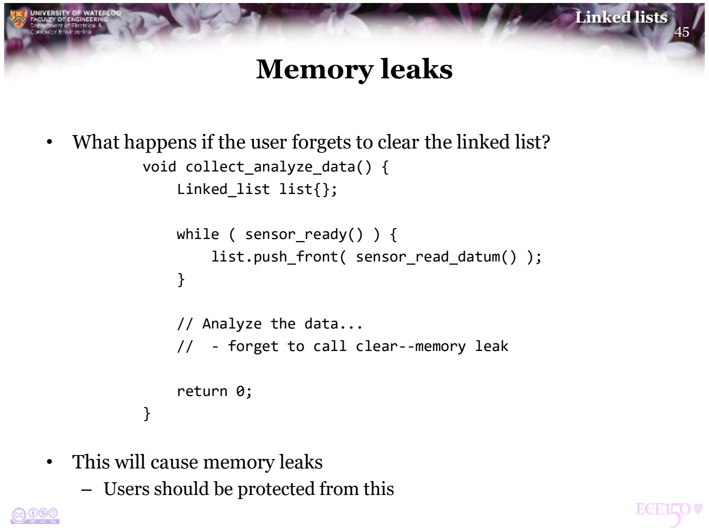
- You could make it more complex, but why bother…



Linked_list::clear()

- Thus, before we exit, we could clear the linked list:
  - This code has no memory leak:

```
int main() {
    Linked_list list{};

    for ( unsigned int k{0}; k <= 100; ++k ) {
        list.push_front( 0.1*k );
    }

    list.clear();

    return 0;
}
```

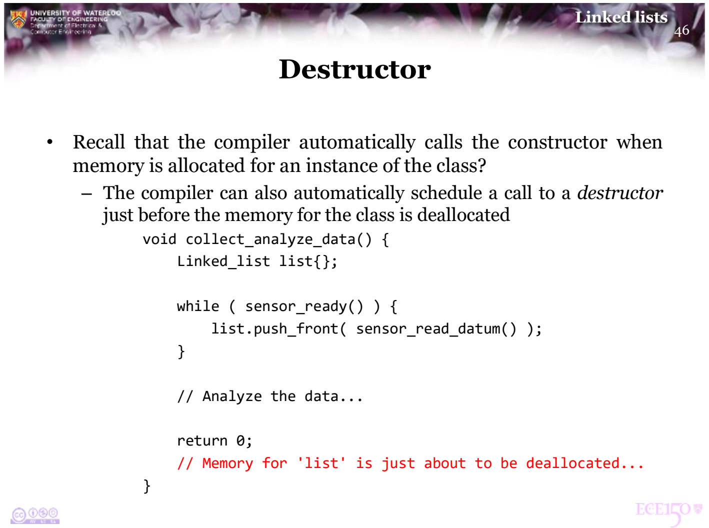


Memory leaks

- What happens if the user forgets to clear the linked list?

```
void collect_analyze_data() {
    Linked_list list{};

    while ( sensor_ready() ) {
        list.push_front( sensor_read_datum() );
    }

    // Analyze the data...
    //  - forget to call clear--memory leak

    return 0;
}
```

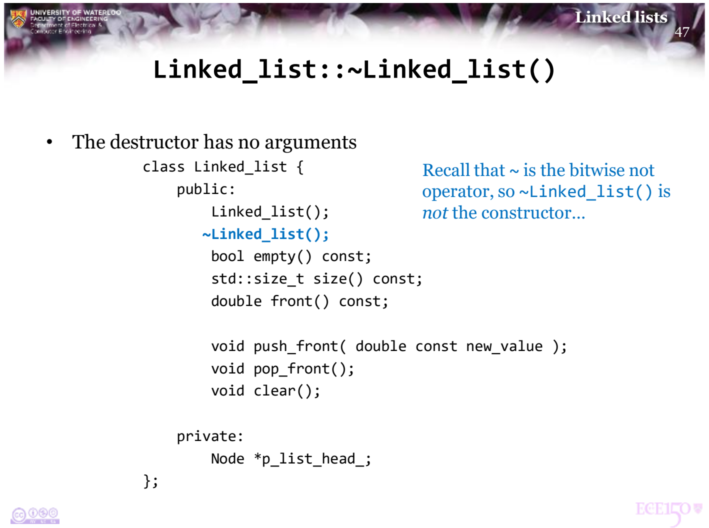
- This will cause memory leaks
  - Users should be protected from this



Destructor

- Recall that the compiler automatically calls the constructor when memory is allocated for an instance of the class?
  - The compiler can also automatically schedule a call to a *destructor* just before the memory for the class is deallocated

```
void collect_analyze_data() {
    Linked_list list{};

    while ( sensor_ready() ) {
        list.push_front( sensor_read_datum() );
    }

    // Analyze the data...

    return 0;
    // Memory for 'list' is just about to be deallocated...
}
```

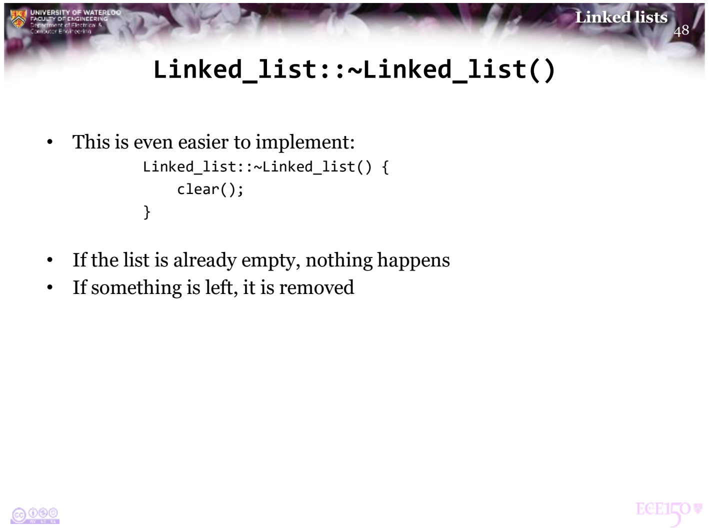


`Linked_list::~Linked_list()`

- The destructor has no arguments

```
class Linked_list {
    public:
        Linked_list();
       ~Linked_list();
        bool empty() const;
        std::size_t size() const;
        double front() const;

        void push_front( double const new_value );
        void pop_front();
        void clear();

    private:
        Node *p_list_head_;
};
```

Recall that ~ is the bitwise not operator, so `~Linked_list()` is *not* the constructor…

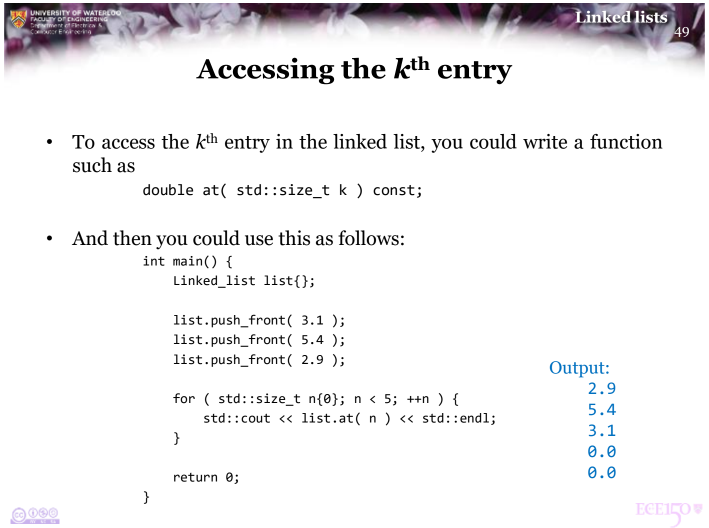


`Linked_list::~Linked_list()`

- This is even easier to implement:

```
Linked_list::~Linked_list() {
    clear();
}
```

- If the list is already empty, nothing happens
- If something is left, it is removed



Accessing the k^{th} entry

- To access the k^{th} entry in the linked list, you could write a function such as

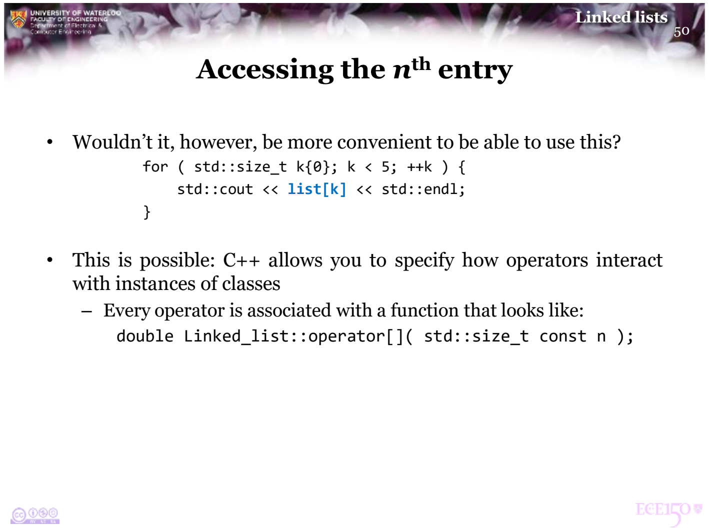
```
double at( std::size_t k ) const;
```

- And then you could use this as follows:

```
int main() {
    Linked_list list{};

    list.push_front( 3.1 );
    list.push_front( 5.4 );
    list.push_front( 2.9 );

    for ( std::size_t n{0}; n < 5; ++n ) {
        std::cout << list.at( n ) << std::endl;
    }

    return 0;
}
```

Output:

```
2.9
5.4
3.1
0.0
0.0
```



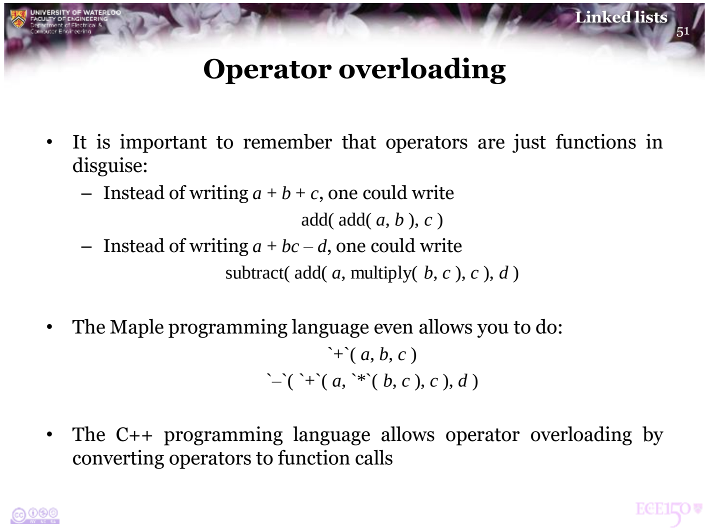
Accessing the n^{th} entry

- Wouldn't it, however, be more convenient to be able to use this?

```
for ( std::size_t k{0}; k < 5; ++k ) {
    std::cout << list[k] << std::endl;
}
```

- This is possible: C++ allows you to specify how operators interact with instances of classes
  - Every operator is associated with a function that looks like:

```
double Linked_list::operator[]( std::size_t const n );
```



Operator overloading

- It is important to remember that operators are just functions in disguise:
  - Instead of writing $a + b + c$, one could write

    add(add(a, b), c)
  - Instead of writing $a + bc - d$, one could write

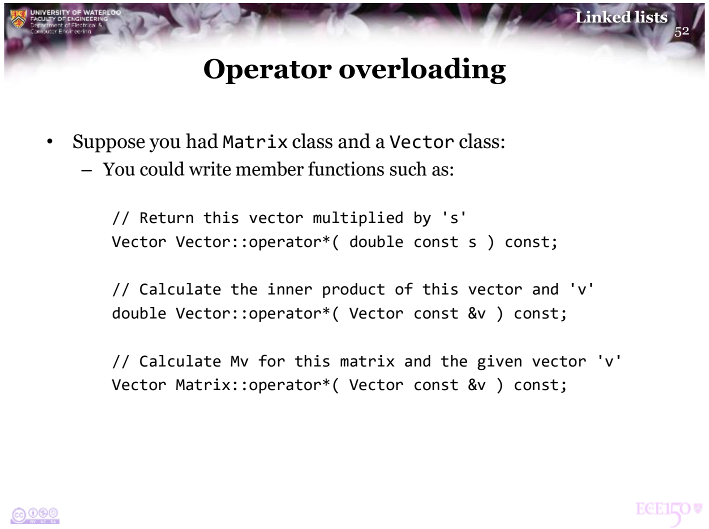
    subtract(add(a, multiply(b, c), c), d)

- The Maple programming language even allows you to do:

  \`+\`(a, b, c)

  \`–\`(\`+\`(a, \`*\`(b, c), c), d)

- The C++ programming language allows operator overloading by converting operators to function calls



Operator overloading

- Suppose you had `Matrix` class and a `Vector` class:
  - You could write member functions such as:

```
// Return this vector multiplied by 's'
Vector Vector::operator*( double const s ) const;

// Calculate the inner product of this vector and 'v'
double Vector::operator*( Vector const &v ) const;

// Calculate Mv for this matrix and the given vector 'v'
Vector Matrix::operator*( Vector const &v ) const;
```

Operator overloading

- In this example, the class definition would have the following function definition:

```
class Linked_list {
    public:
        bool empty() const;
        std::size_t size() const;
        double front() const;
        double operator[]( std::size_t const n ) const;

        // Other member functions
    private:
        // The private member variables
};
```

Operator overloading

- The member function definition is identical to that of member functions:

```
double Linked_list::operator[]( std::size_t const n ) const {
    std::size_t k{0};
    Node *p_current_node{ p_list_head_ };

    while ( p_current_node != nullptr ) {
        if ( k == n ) {
            return p_current_node->value_;
        }

        ++k;
        p_current_node = p_current_node->p_next_node_;
    }

   // We are beyond the end of the linked list: return 0.0
    return 0.0;
}
```

Operator overloading

- The following two are consequently equivalent:

```
for ( std::size_t n{0}; n < 5; ++n ) {
    std::cout << list[n] << std::endl;
    std::cout << list.operator[]( n ) << std::endl;
}
```

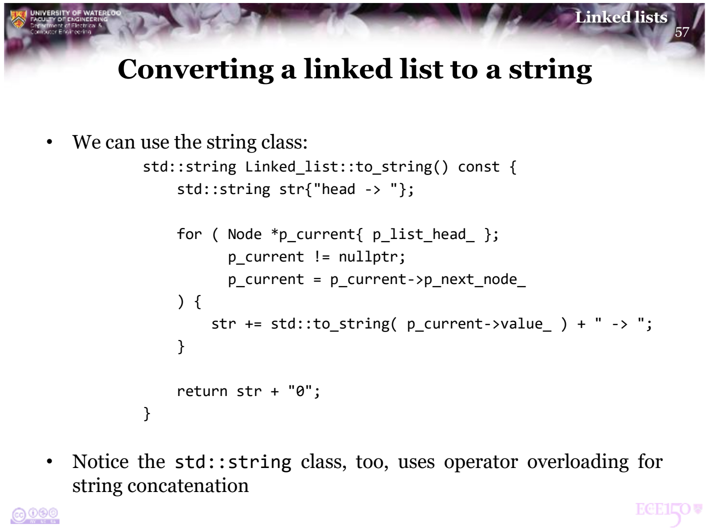
- The compiler converts the `list[n]` into the corresponding function call for you
  - You don't have to do anything

```
double Linked_list::operator[]( std::size_t const n ) const {
    double k{0};
    Node *p_current_node{ p_list_head_ };

    while ( p_current_node != nullptr ) {
        if ( k == n ) {
            return p_current_node->value_;
        }

        ++k;
        p_current_node = p_current_node->p_next_node_;
    }

   // We are beyond the end of the linked list: return 0.0
    return 0.0;
}
```

Converting a linked list to a string

- Suppose we want to convert a linked list to a string
  - Such a string could be printed

```
class Linked_list {
    public:
        bool empty() const;
        std::size_t size() const;
        double front() const;
        std::string to_string() const;
        double operator[]( std::size_t const n ) const;

        // Other member functions
    private:
        // The private member variables
};
```

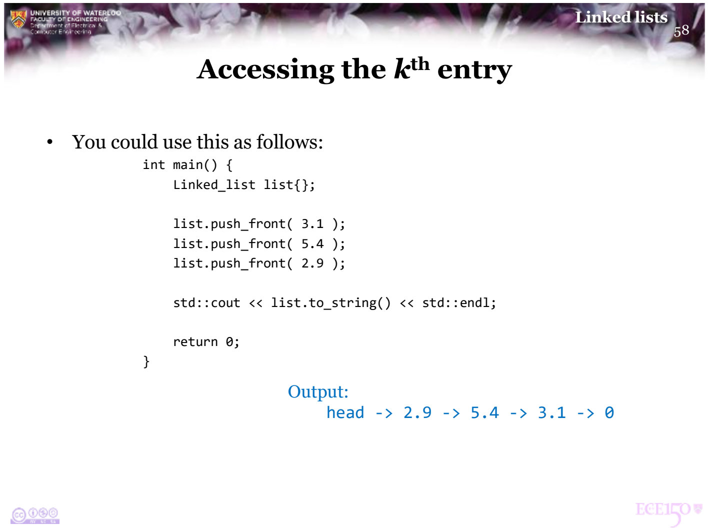


Converting a linked list to a string

- We can use the string class:

```
std::string Linked_list::to_string() const {
    std::string str{"head -> "};

    for ( Node *p_current{ p_list_head_ };
          p_current != nullptr;
          p_current = p_current->p_next_node_
    ) {
        str += std::to_string( p_current->value_ ) + " -> ";
    }

    return str + "0";
}
```

- Notice the `std::string` class, too, uses operator overloading for string concatenation



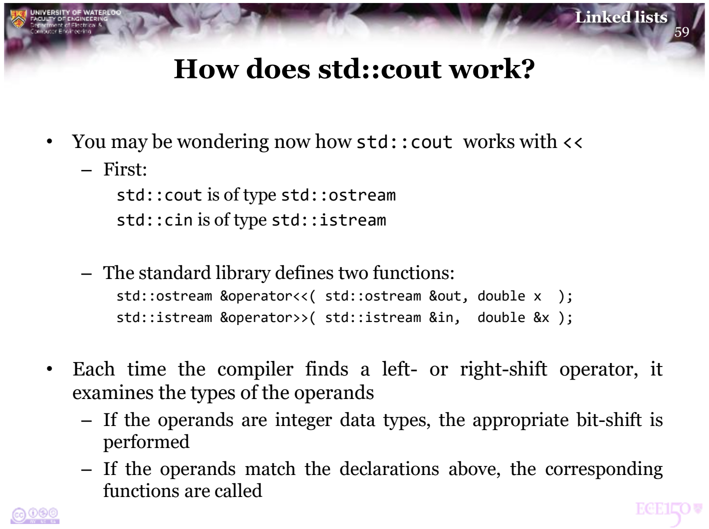
Accessing the k^{th} entry

- You could use this as follows:

```
int main() {
    Linked_list list{};

    list.push_front( 3.1 );
    list.push_front( 5.4 );
    list.push_front( 2.9 );

    std::cout << list.to_string() << std::endl;

    return 0;
}
```

Output:

```
head -> 2.9 -> 5.4 -> 3.1 -> 0
```



How does std::cout work?

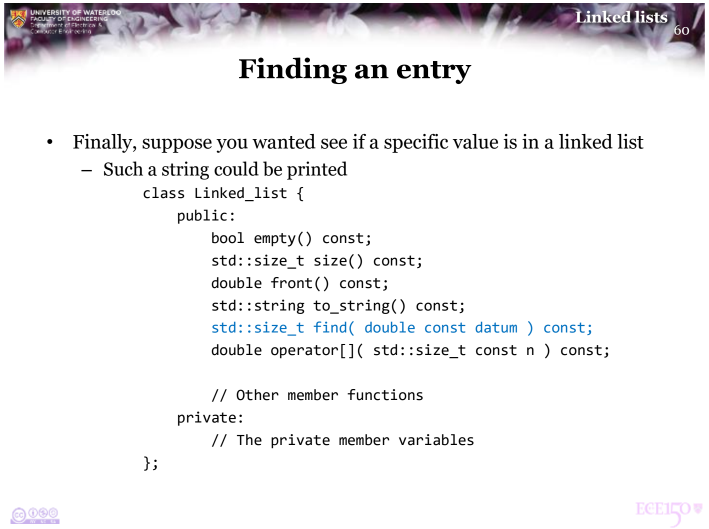
- You may be wondering now how `std::cout` works with <<
  - First:

    `std::cout` is of type `std::ostream`

    `std::cin` is of type `std::istream`
  - The standard library defines two functions:

```
std::ostream &operator<<( std::ostream &out, double x  );
std::istream &operator>>( std::istream &in,  double &x );
```

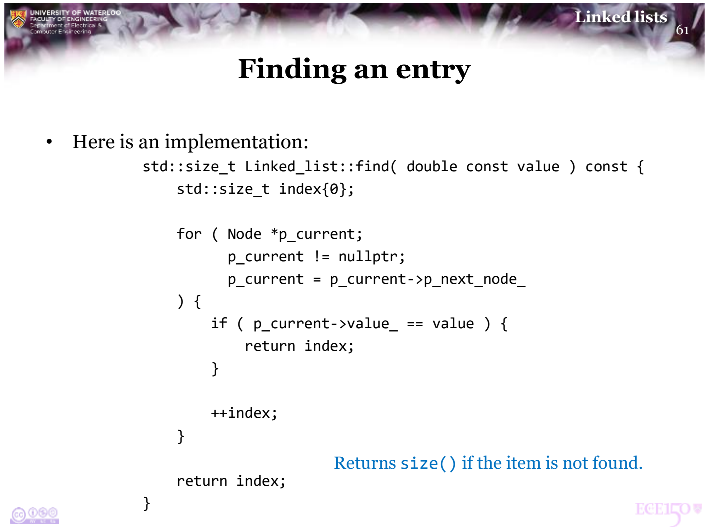
- Each time the compiler finds a left- or right-shift operator, it examines the types of the operands
  - If the operands are integer data types, the appropriate bit-shift is performed
  - If the operands match the declarations above, the corresponding functions are called



Finding an entry

- Finally, suppose you wanted see if a specific value is in a linked list
  - Such a string could be printed

```
class Linked_list {
    public:
        bool empty() const;
        std::size_t size() const;
        double front() const;
        std::string to_string() const;
        std::size_t find( double const datum ) const;
        double operator[]( std::size_t const n ) const;

        // Other member functions
    private:
        // The private member variables
};
```

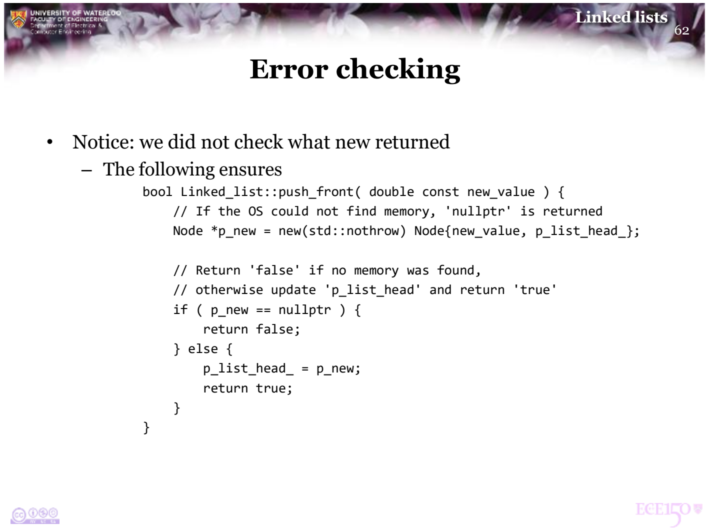


Finding an entry

- Here is an implementation:

```
std::size_t Linked_list::find( double const value ) const {
    std::size_t index{0};

    for ( Node *p_current;
          p_current != nullptr;
          p_current = p_current->p_next_node_
    ) {
        if ( p_current->value_ == value ) {
            return index;
        }

        ++index;
    }

    return index;
}
```

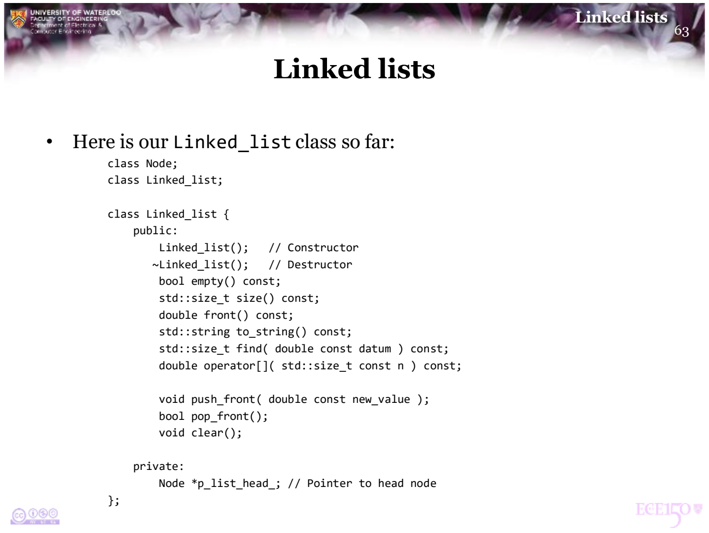
Returns `size()` if the item is not found.



Error checking

- Notice: we did not check what new returned
  - The following ensures

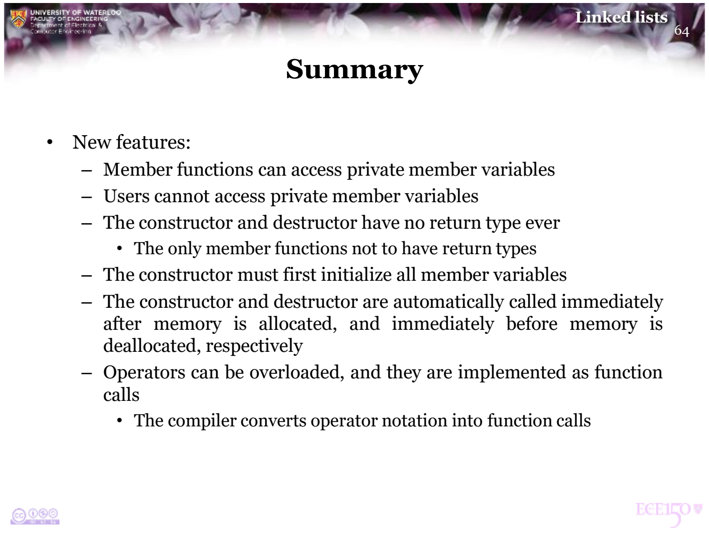
```
bool Linked_list::push_front( double const new_value ) {
    // If the OS could not find memory, 'nullptr' is returned
    Node *p_new = new(std::nothrow) Node{new_value, p_list_head_};

    // Return 'false' if no memory was found,
    // otherwise update 'p_list_head' and return 'true'
    if ( p_new == nullptr ) {
        return false;
    } else {
        p_list_head_ = p_new;
        return true;
    }
}
```



Linked lists

- Here is our `Linked_list` class so far:

```
class Node;
class Linked_list;

class Linked_list {
    public:
        Linked_list();   // Constructor
       ~Linked_list();   // Destructor
        bool empty() const;
        std::size_t size() const;
        double front() const;
        std::string to_string() const;
        std::size_t find( double const datum ) const;
        double operator[]( std::size_t const n ) const;

        void push_front( double const new_value );
        bool pop_front();
        void clear();

    private:
        Node *p_list_head_; // Pointer to head node
};
```



Summary

- New features:
  - Member functions can access private member variables
  - Users cannot access private member variables
  - The constructor and destructor have no return type ever
    - The only member functions not to have return types
  - The constructor must first initialize all member variables
  - The constructor and destructor are automatically called immediately after memory is allocated, and immediately before memory is deallocated, respectively
  - Operators can be overloaded, and they are implemented as function calls
    - The compiler converts operator notation into function calls

Summary

- Following this lesson, you now
  - Know how to create a simple linked list class
  - Understand how member functions can implement most of the operations in question
  - Know how to step through a linked list

References

[1] No references?

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.