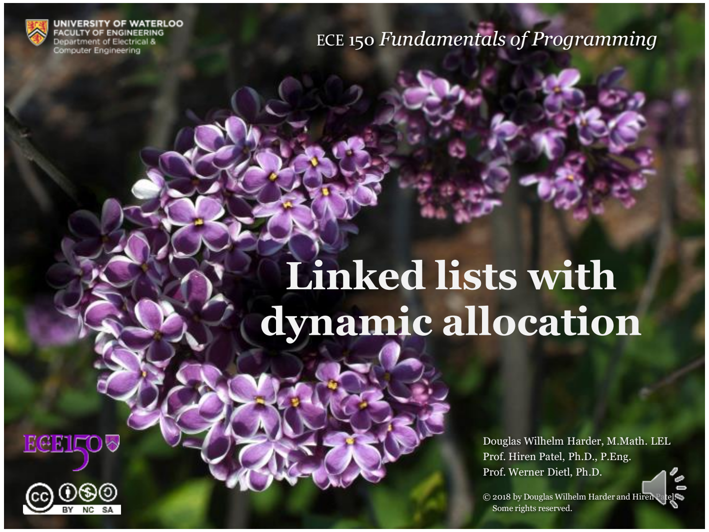


1

Outline

- In this lesson, we will:
  - Describe how to create a linked list using addresses
  - Learn how to add, access and remove nodes from such a linked list
  - See how to ensure that we do not have any memory leaks
  - Learn how to loop through a linked list
  - See how friendship can be used to access private member variables when necessary

2

Nodes

- In the last topic, we introduced the class:

```
class Node {
    public:
        double      value_;
        std::size_t next_index_;
};
```

- In our linked list in that topic:
  - All nodes were stored in an array
  - Each node stored a value and the index of the next node
  - We required one local variable to store the index of the first node

3

Nodes

- Suppose, instead, we dynamically allocated memory each time we required a new node
  - Each new node will be allocated using a call to `new Node{…};`
  - Consequently, we will be able to associate the node not with an index, but with an address
- Consequently, we will be storing the *address* of the next node:

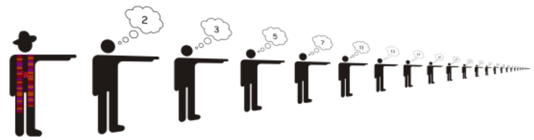
```
class Node {
    private:
        double value_;
        Node   *p_next_node_;
};
```

4

Nodes

- Thus, with a local variable storing the address of the first node:

```
Node *p_list_head{ nullptr };
```



Nodes

- In a sense, these two aren't very different:
  - In the first, you have an array of possible nodes you could use
  - In the second, all of memory is essentially an array and the address is an index into that array

```
class Node {
    public:
        double      value_;
        std::size_t next_index_;
};

class Node {
    private:
        double value_;
        Node  *p_next_node_;
};
```


Nodes

- Let us introduce the necessary public member functions:

```
class Node {
    public:
        Node( double const  new_value,
              Node   const *p_new_next_node );
        Node( Node const &original ) = delete;
        Node( Node       &&original ) = delete;
        Node &operator=( Node const &rhs ) = delete;
        Node &operator=( Node       &&rhs ) = delete;

        double value() const;
        Node *p_next_node() const;
        void value( double const new_value );
        void p_next_node( Node *const new_p_next_node );

    private:
        double value_;
        Node  *p_next_node_;
};
```


Nodes

- Let us introduce the necessary public member functions:

```
Node::Node( double const new_value,
            Node  *const p_new_next_node ):
value_{ new_value },
p_next_node_{ p_new_next_node } {
    // Empty constructor
}
double Node::value() const {
    return value_;
}
Node *Node::p_next_node() const {
    return p_next_node_;
}
void Node::value( double const new_value ) {
    value_ = new_value;
}
void Node::p_next_node( Node *const p_new_next_node ) {
    p_next_node_ = p_new_next_node;
}
```

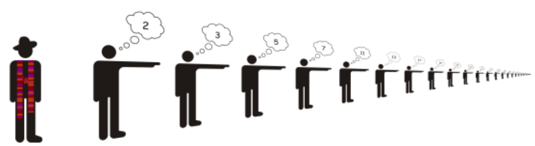
Nodes

- For a linked list, we also need to store the address of the first node

```
Node *p_list_head{ nullptr };
```

  – We can then add new nodes to this linked list
- Unfortunately, the user may accidentally assign this

```
p_list_head = nullptr;
```

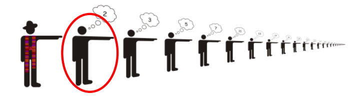


Nodes

- Let us also introduce a linked list class:
  – It will store the address of the first node in the linked list

```
class Linked_list {
    public:
        Linked_list();
       ~Linked_list();
        Linked_list( Linked_list const &original ) = delete;
        Linked_list( Linked_list      &&original ) = delete;
        Linked_list &operator=(Linked_list const &rhs ) = delete;
        Linked_list &operator=(Linked_list      &&rhs ) = delete;

        double front() const;
        void push_front( double new_value );
    private:
        Node *p_list_head_;
};
```



Nodes

- An initial linked list is empty,
  so the list head will be assigned the null pointer

```
Linked_list::Linked_list():
p_list_head_{ nullptr } {
    // Empty constructor
}

Linked_list::~Linked_list() {
    // We are using dynamic memory allocation,
    //     so we will have to implement this...
}
```


Push front

- Now, suppose we start with an empty linked list:

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

    data.push_front( 4.2 );

    return 0;
}
```

Linked lists with dynamic memory allocation 13

Push front when empty

- When we go from no nodes to one,
  initially the linked list is pointing to the null pointer
  - Once we have initialized the first node,
    the linked list is pointing to the first node

13

Linked lists with dynamic memory allocation 14

Push front when empty

- This is what we start with:

```
0xffffff0 | 0x0      | p_list_head_      0xffffff0 | 0x303030 | p_list_head_

                                         0x303030  |   4.2    | value_
                                                   | 0x0      | p_next_node_
```

  - How do we get here?

```
void Linked_list::push_front( double new_value ) {
    if ( p_list_head_ == nullptr ) {
        Node *p_new_node{ new Node{ new_value, nullptr } };
        p_list_head_ = p_new_node;
    } else {
        // ...
    }
}
```

14

Linked lists with dynamic memory allocation 15

Push front when not empty

- Now, let's add one more value at the front of this linked list:

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

    data.push_front( 4.2 );
    data.push_front( 9.1 );

    return 0;
}
```

15

Linked lists with dynamic memory allocation 16

Push front when not empty

- When there are one or more nodes in the linked list,
  we are just adding another node to the front
  - It doesn't matter if we started with one
    or a thousand nodes in the linked list

16

Push front when not empty

- This is what we start with:

0xfffff0	0x303030	p_list_head_
0x303030	4.2	value_
	0x0	p_next_node_

0xfffff0	0xbdbdb8	p_list_head_
0x303030	4.2	value_
	0x0	p_next_node_
0xbdbdb8	9.1	value_
	0x303030	p_next_node_

  – How do we get here?

```
void Linked_list::push_front( double new_value ) {
    if ( p_list_head_ == nullptr ) {
        Node *p_new_node{ new Node{ new_value, nullptr } };
        p_list_head_ = p_new_node;
    } else {
        Node *p_new_node{ new Node{ new_value, p_list_head_ } };
        p_list_head_ = p_new_node;
    }
}
```


Simplifying push front

- Can we simplify this in any way?
  – If you ever see:

```
double new_x{ 3.2 };
y = new_x;
```

  – You can always replace this with:

```
y = 3.2;
```

- Here we have:

```
Node *p_new_node{ new Node{ new_value, nullptr } };
p_list_head_ = p_new_node;
```

  – You can always replace this with:

```
p_list_head_ = new Node{ new_value, nullptr };
```


Simplifying push front

- Now we have:

```
void Linked_list::push_front( double new_value ) {
    if ( p_list_head_ == nullptr ) {
        Node *p_new_node{ new Node{ new_value, nullptr } };
        p_list_head_ = p_new_node;
    } else {
        Node *p_new_node{ new Node{ new_value, p_list_head_ } };
        p_list_head_ = p_new_node;
    }
}

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


Simplifying push front

- Next, note that if `p_list_head_ == nullptr`, can't we just use that as the second argument in the constructor?

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

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

Simplifying push front

- Next, the consequent and alternative blocks are now identical:

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


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

21

The first value (front)

- How do we return the value stored in the first node?

```
double Linked_list::front() const {
    if ( p_list_head_ != nullptr ) {
        return p_list_head_->value();
    } else {
        assert( p_list_head_ == nullptr );
        throw std::out_of_range{ "The linked list is empty" };
    }
}
```

22

Empty?

- How does the user know if the linked list is empty?

```
class Linked_list {
    public:
        // Constructors, copy constructors, destructor, etc.
        bool empty() const;
    private:
        Node *p_list_head_;
};

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

23

Empty?

- Some may write this as the following:

```
bool Linked_list::empty() const {
    if ( p_list_head_ == nullptr ) {
        return true;
    } else {
        return false;
    }
}
```

- This is, however, equivalent to:

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

24

Popping the front node

- Suppose we want to remove the first node in the linked list:

```
class Linked_list {
    public:
        // Constructors, copy constructors, destructor, etc.
        void pop_front();
    private:
        Node *p_list_head_;
};
```

- As an aside:
  - All the member function identifiers and their behaviors are very close to the corresponding member functions of classes in the Standard Template Library (the STL)
    - Consequently, after this course, you should already be familiar with how to use some of the STL classes

ECE150

25

Popping the front node

- Suppose we want to remove the front node
  - We cannot do this if the linked list is empty

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        // Remove the front node
    }
}
```

  - Notice we changed

```
        if ( p_list_head_ != nullptr ) {
```

    to

```
        if ( !empty() ) {
```

  - Benefits of this approach include:
    - Easier to read
    - If the implementation changes, `empty()` still works

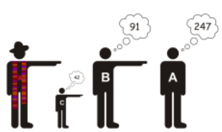
ECE150

26

Popping the front when not empty

- Suppose we want to remove the front of this linked list

  - To update the linked list, just point the head to whatever the previous first node was pointing to



ECE150

27

Popping the front when not empty

- Thus, we can remove the first node as follows:

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        p_list_head_ = p_list_head_->p_next_node();
    }
}
```

ECE150

28

Popping the front when not empty

- Thus, starting with this linked list with three nodes:

0xfffff0	0xbdbdb8	p_list_head_
0x303030	4.2	value_
	0x0	p_next_node_
0xbdbdb8	9.1	value_
	0x303030	p_next_node_
0x1f1f18	6.5	value_
	0xbdbdb8	p_next_node_

```
p_list_head_ = p_list_head_->p_next_node();
```

  – What is the problem here?

29

Popping the front when not empty

- Does this fix the problem?

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        delete p_list_head_;
        p_list_head_ = p_list_head_->p_next_node();
    }
}
```

30

Popping the front when not empty

- Again, starting with this linked list with three nodes:

0xfffff0	0x1f1f18	p_list_head_
0x303030	4.2	value_
	0x0	p_next_node_
0xbdbdb8	9.1	value_
	0x303030	p_next_node_
0x1f1f18	6.5	value_
	0xbdbdb8	p_next_node_

```
delete p_list_head_;
p_list_head_ = p_list_head_->p_next_node();
```

31

Popping the front when not empty

- What we really require is a temporary pointer to this node:

0xfffff0	0xbdbdb8	p_list_head_
0x303030	4.2	value_
	0x0	p_next_node_
0xbdbdb8	9.1	value_
	0x303030	p_next_node_
0x1f1f18	6.5	value_
	0xbdbdb8	p_next_node_

32

Popping the front when not empty

- Thus, we could pop the front node as follow:

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

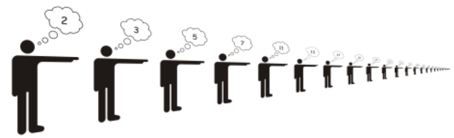

Destructor

- What happens when the linked list variable goes out of scope?

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

    data.push_front( 4.2 );
    data.push_front( 9.1 );
    data.push_front( 6.5 );

    return 0;
}
```



Destructor

- What happens when the linked list variable goes out of scope?

0xffffff0	0x1f1f18	p_list_head_
0x303030	4.2	value_
	0x0	p_next_node_
0xbdbdb8	9.1	value_
	0x303030	p_next_node_
0x1f1f18	6.5	value_
	0xbdbdb8	p_next_node_

Destructor

- To avoid the memory leak,
  it is necessary to empty the linked list first

0xffffff0	0x1f1f18	p_list_head_
0x303030	4.2	value_
	0x0	p_next_node_
0xbdbdb8	9.1	value_
	0x303030	p_next_node_
0x1f1f18	6.5	value_
	0xbdbdb8	p_next_node_

Linked lists with dynamic memory allocation 37

Destructor

- There are easy ways of doing this, and hard ways
  - The easy is to use what you already have:

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

37

Linked lists with dynamic memory allocation 38

Linked list class

- This is our linked list class so far:

```
class Linked_list {
    public:
        Linked_list();
       ~Linked_list();
        Linked_list( Linked_list const &original ) = delete;
        Linked_list( Linked_list      &&original ) = delete;
        Linked_list &operator=(Linked_list const &rhs ) = delete;
        Linked_list &operator=(Linked_list      &&rhs ) = delete;

        double front() const;
        bool empty() const;
        void push_front( double new_value );
        void pop_front();
    private:
        Node *p_list_head_;
};
```

38

Linked lists with dynamic memory allocation 39

Linked list class

- What else could be implemented?

```
std::size_t size() const;
bool operator==( Linked_list const &rhs ) const;
bool operator!=( Linked_list const &rhs ) const;
std::size_t find( double const value ) const;

double back() const;
void pop_back();

void clear();

std::size_t erase( std::size_t const index );
std::size_t erase( std::size_t const first_index,
                   std::size_t const  last_index );

void reverse();
```

39

Linked lists with dynamic memory allocation 40

Clearing a linked list

- The easiest to implement is the clear member function:

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

- We can now simplify the destructor:

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

40

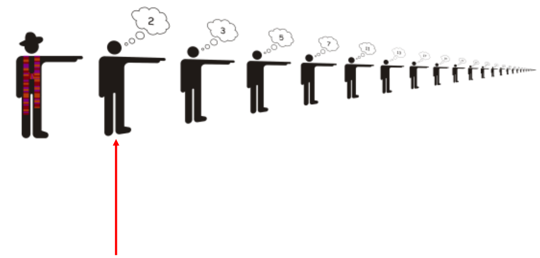
Looping through a linked list

- All the additional member functions we've suggested require us to walk through the nodes in the linked list
  - We will now go through the process of looping through a linked list
  - All the additional member functions suggested will require such a process of looping through the nodes in order to
    - Access the necessary information for the correct return value
    - Performing the appropriate modification to the linked list

41

Looping through a linked list

- How would you step through this linked list?



42

Looping through a linked list

- How can we loop through the linked list node by node?

```
Node *p_node{ p_list_head_ };

while ( p_node != nullptr ) {
    // Do something...

    p_node = p_node->p_next_node();
}
```

0x303030	4.2	value_
	0x0	p_next_node_
0xbdbdb8	9.1	value_
	0x303030	p_next_node_
0x1f1f18	6.5	value_
	0xbdbdb8	p_next_node_
0xffffff0	0x1f1f18	p_list_head_

43

Looping through a linked list

- We can rewrite this while loop as a for loop

```
for ( Node *p_node{ p_list_head_ }; p_node != nullptr;
                                   p_node = p_node->p_next_node() ) {
    // Access or modify the current node with
    //       p_node->value()
    //       p_node->p_next_node()
    //       p_node->value( … );
    //       p_node->p_next_node( … );
    if ( p_node->p_next_node() != nullptr ) {
        // Access or modify the following node with
        //       p_node->p_next_node()->value()
        //       p_node->p_next_node()->p_next_node()
        //       p_node->p_next_node()->value( … );
        //       p_node->p_next_node()->p_next_node( … );
    }
}
```

44

Size?

- How do we count how many items are in the linked list?

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

    for ( Node *p_node{ p_list_head_ };
           p_node != nullptr; p_node = p_node->p_next_node() ) {
        ++list_size;
    }

    return list_size;
}
```


Erase?

- How do we erase the k^{th} entry in the linked list?

```
std::size_t Linked_list::erase( std::size_t index ) {
    if ( empty() ) {
        return 0;
    } else if ( index == 0 ) {
        pop_front();
        return 1;
    } else {
        // We must loop through the linked list,
        // but we must modify the node immediately before the
        // node we want to erase
        //  - How do we get to this particular node?
        //  - What changes must be made to erase the next node?
        //  - How do we determine if there are an insufficient
        //    number of nodes in the linked list?
    }
}
```


Printing

- Can we print a linked list?

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

    data.push_front( 4.2 );
    data.push_front( 9.1 );
    data.push_front( 6.5 );

    std::cout << data << std::endl;

    return 0;
}
```

Output:

```
head -> 6.5 -> 9.1 -> 4.2 -> nullptr
(6.5, 9.1, 4.2)
```


Printing

- We have already seen how to overload `operator<<` to print

```
std::ostream &operator<<( std::ostream &out,
                          Linked_list const &list ) {
    out << "head -> ";

    for ( Node *p_node{ list.p_list_head_ };
            p_node != nullptr; p_node = p_node->p_next_node() ) {
        out << p_node->value() << " -> ";
    }

    out << "nullptr";

    return out;
}
```

Printing

- Problem:
  – The `p_list_head_` member variable is private!

```
std::ostream &operator<<( std::ostream &out,
                          Linked_list const &list ) {
    out << "head -> ";

    for ( Node *p_node{ list.p_list_head_ };
            p_node != nullptr; p_node = p_node->p_next_node() ) {
        out << p_node->value() << " -> ";
    }

    out << "nullptr";

    return out;
}
```


Printing

- C++ allows a class to specify functions and entire classes that are allowed access to all private members

```
// Class declaration
class Linked_list;
// Function declaration
std::ostream &operator<<( std::ostream &out,
                          Linked_list const &list );
// Class definition
class Linked_list {
    public:

    private:

    friend std::ostream &operator<<( std::ostream      &out,
                                     Linked_list const &list );
};
// Function definitions...
```


Summary

- Following this lesson, you now
  – Know how to use addresses to create a linked list
  – Understand why it is also necessary to create a linked list class
  – Know how to add new nodes to the front of a linked list
  – Know how to access that first node
  – Know how to remove, or pop, that first node
  – Know how to easily implement the destructor
  – Understand how to loop through the nodes of a linked list
  – Understand the idea of friendship in C++

References

[1] https://en.wikipedia.org/wiki/Linked_list
[2] https://en.wikipedia.org/wiki/Node_(computer_science)

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.

53

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.

54