Outline

- In this lesson, we will:
  - Review that arrays can be great under certain circumstances
  - See that there are significant weaknesses
    - Fixed size
    - Expensive to move many entries
    - Same type
  - Understand that there must be other approaches

Outline

- How do we solve the previous problem?
  - A linked list stored inside an array has fixed size…
- Solution:
  - Ask the operating system for a new node
- How do we reference such a node?
  - An address instead of an index…

Nodes

- Here is a node class:

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

- The second member variable stores the *address* of the next node
  - What is the default value for the end of the list?
  - Answer: `nullptr`

Dynamically Allocated Nodes

- Accessing member variables of dynamically allocated Node:

```
Node *p_node{ new Node{99.9, nullptr} };

// Normally, we would do this:
std:cout << (*p_node).value_ << std::endl;

// This is equivalent:
//  - access member variables using -> operator
//  - the left-hand operand must be a pointer to an object
//    with the right-hand being a member variable
std::cout << p_node->value_ << std::endl;
```

- The -> is a syntactical short form to access member of objects via a pointer to the object

Nodes

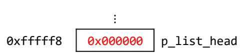
- We can now create a linked list:

```
int main() {
    // Create an empty linked list
    //  - the list head stores the null pointer
    Node *p_list_head{nullptr};

    // Put one item in the linked list
    p_list_head = new Node{4.2, nullptr};

    // Append a second item at the end of this linked list
    p_list_head->p_next_node_ = new Node{9.9, nullptr};

    // Prepend a new item at the start of this linked list
    p_list_head = new Node{1.3, p_list_head};

    return 0;
}
```


Nodes

- Let's step through this one step at a time:

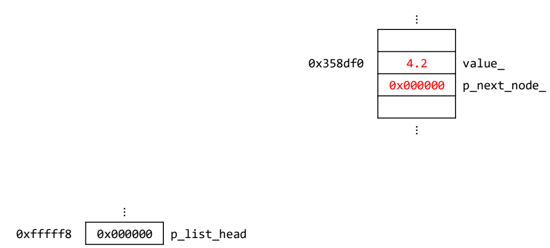
```
Node *p_list_head{nullptr};
```



Nodes

- Add a node to this list: create a new node and initialize it

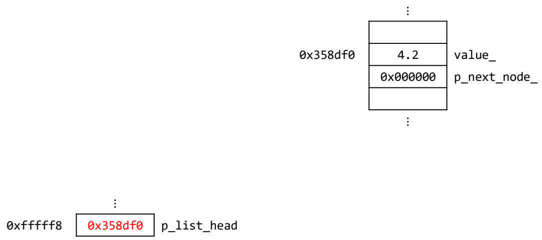
```
p_list_head = new Node{4.2, nullptr};
```



Nodes

- Add a node to this list: assign the address to `p_list_head`

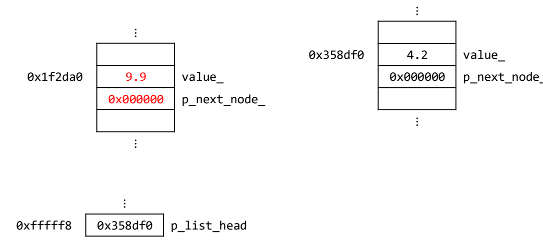
```
p_list_head = new Node{4.2, nullptr};
```



Nodes

- Append a node after this node

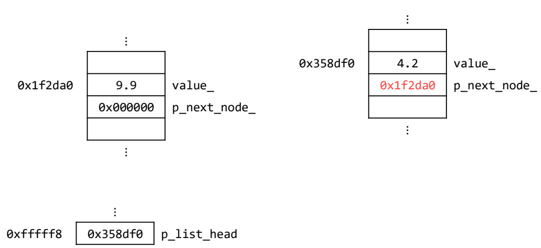
```
p_list_head->p_next_node_ = new Node{9.9, nullptr};
```



Nodes

- Assign this returned address to `p_list_head->p_next_node_`

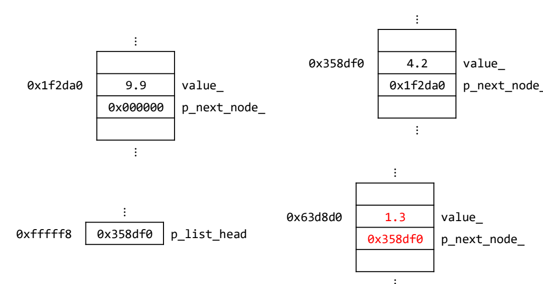
```
p_list_head->p_next_node_ = new Node{9.9, nullptr};
```



Nodes

- Prepend a value to the start of the linked list

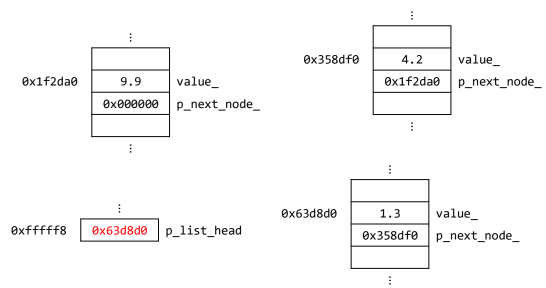
```
p_list_head = new Node{1.3, p_list_head};
```



Nodes

- Assign the address of this to `p_list_head`:

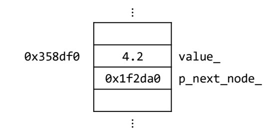
```
p_list_head = new Node{1.3, p_list_head};
```

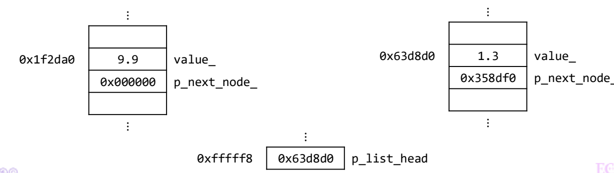


Nodes

- Let's count how many items there are in this linked list:

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

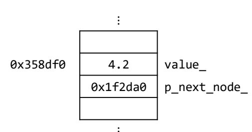


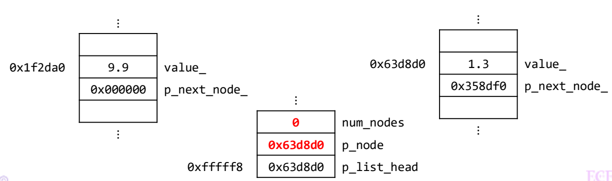


Nodes

- Initialize two new local variables:

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

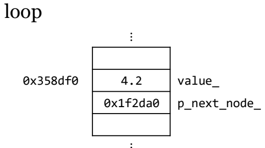


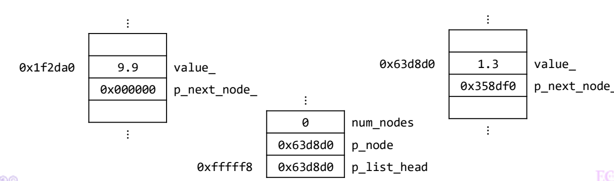


Nodes

- The condition is true, so we execute the loop

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

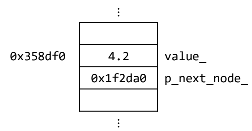


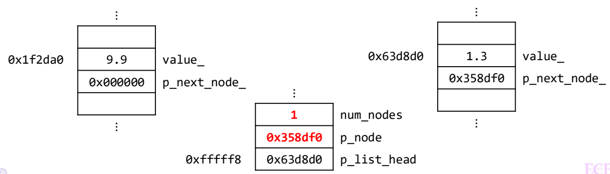


Nodes

- We update the two local variables:

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

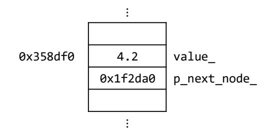


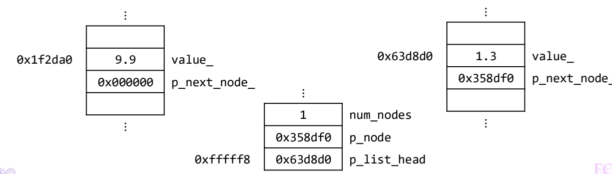


Nodes

- The condition is true, so we execute the loop

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

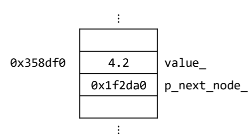


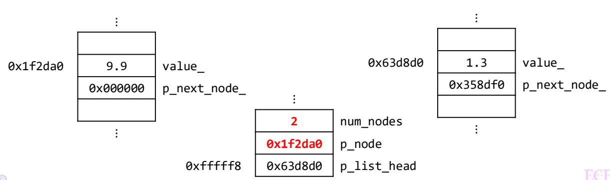


Nodes

- We update the two local variables:

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

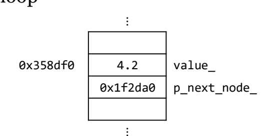


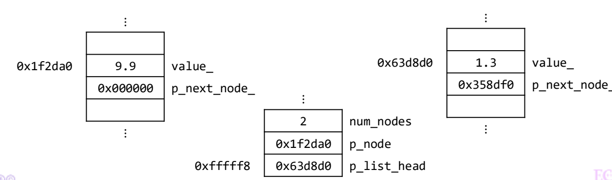


Nodes

- The condition is true, so we execute the loop

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

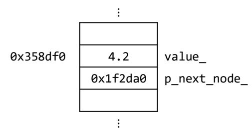


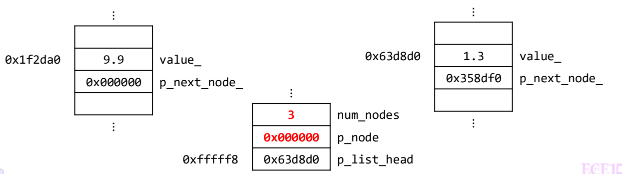


Nodes

- We update the two local variables:

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

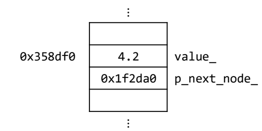


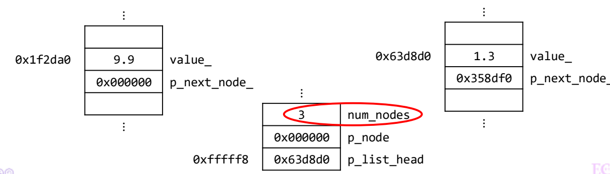


Nodes

- The condition is false, so we are finished and we have the size

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```

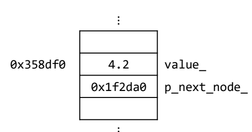


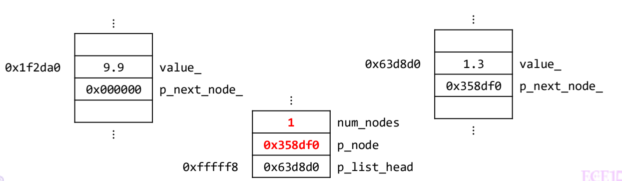


Nodes

- We update the two local variables:

```
Node *p_node{p_list_head};
unsigned int num_nodes{0};

while ( p_node != nullptr ) {
    p_node = p_node->p_next_node_;
    ++num_nodes;
}
```



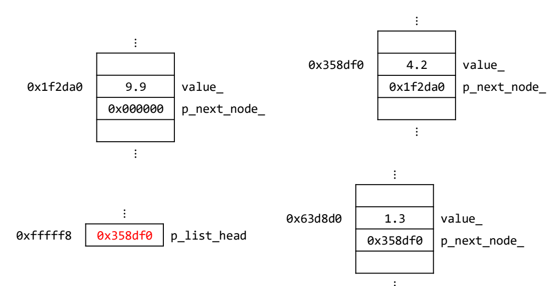


Deleting the first node

- How do we remove the first node from the linked list?
  - What's wrong with

```
p_list_head = p_list_head->p_next_node_;
```

Memory leak!

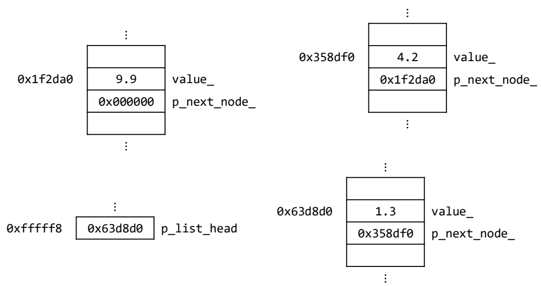


Deleting the first node

- We must delete the node to avoid a memory leak, so how about:

```
delete p_list_head;
p_list_head = p_list_head->p_next_node_;
```

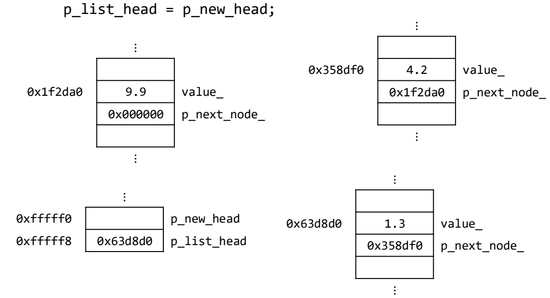
Dangling pointer!



Deleting the first node

- Save the current head, delete the first node, and then update

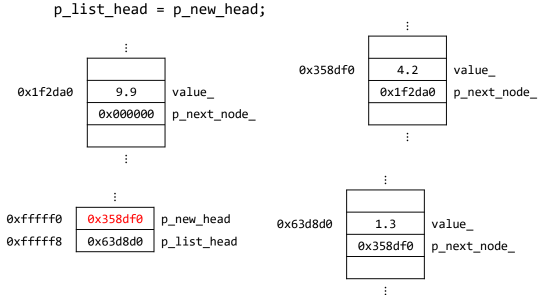
```
Node* p_new_head{p_list_head->p_next_node_};
delete p_list_head;
p_list_head = p_new_head;
```



Deleting the first node

- Save the current head, delete the first node, and then update

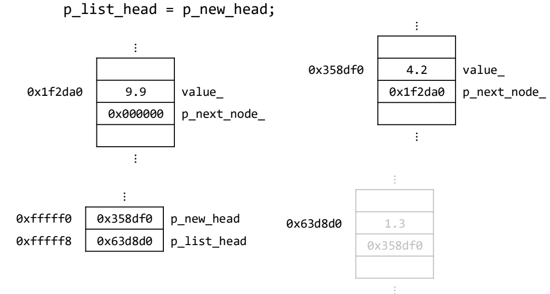
```
Node* p_new_head{p_list_head->p_next_node_};
delete p_list_head;
p_list_head = p_new_head;
```



Deleting the first node

- Save the current head, delete the first node, and then update

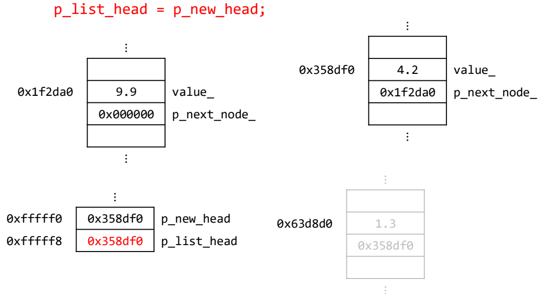
```
Node* p_new_head{p_list_head->p_next_node_};
delete p_list_head;
p_list_head = p_new_head;
```



Deleting the first node

- Save the current head, delete the first node, and then update

```
Node* p_new_head{p_list_head->p_next_node_};
delete p_list_head;
p_list_head = p_new_head;
```



Appending a node

- How do we know if a linked list is empty?
  - If the list head stores equals the null pointer:

```
bool is_empty( Node *p_list_head ) {
    return ( p_list_head == nullptr );
}
```


Appending a node

- How do we append a new node?
  - Find the last node
  - Update it to store the address of a new node that:
    - Contains the new value
    - Has a null pointer as its next pointer

Appending a node

- Finding the last node sounds like a function:

```
Node *list_tail( Node *p_list_head ) {
    Node *p_list_tail{p_list_head};

    while ( p_list_tail->p_next_node_ != nullptr ) {
        p_list_tail = p_list_tail->p_next_node_;
    }

    return p_list_tail;
}
```

Appending a node

- What happens if the list is empty?

```
Node *list_tail( Node *p_list_head ) {
    if ( is_empty( p_list_head ) ) {
        return nullptr;
    }

    Node *p_list_tail{p_list_head};

    while ( p_list_tail->p_next_node_ != nullptr ) {
        p_list_tail = p_list_tail->p_next_node_;
    }

    return p_list_tail;
}
```


Appending a node

- We now append a new node to the end of the linked list:

```
void push_back( Node *p_list_head,
                double const new_value ) {
    Node *p_list_tail{list_tail( p_list_head )};

    p_list_tail->p_next_node_
         = new Node{ new_value, nullptr };
}
```


Appending a node

- Does this fix the problem if the list is empty?

```
void push_back( Node *p_list_head,
                double const new_value ) {
    if ( is_empty( p_list_head ) ) {
        p_list_head = new Node{ new_value, nullptr };
    } else {
        Node *p_list_tail{list_tail( p_list_head )};

        p_list_tail->p_next_node_
            = new Node{ new_value, nullptr };
    }
}
```


Appending a node

- We must pass the list head by reference:

```
void push_back( Node *&p_list_head,
                double const new_value ) {
    if ( is_empty( p_list_head ) ) {
        p_list_head = new Node{ new_value, nullptr };
    } else {
        Node *p_list_tail{list_tail( p_list_head )};

        p_list_tail->p_next_node_
            = new Node{ new_value, nullptr };
    }
}
```

Going out of scope…

- Suppose we call a function which creates a list:

```
void function_name() {
    Node *p_my_list{nullptr};

    push_back( p_my_list, 4.7 );
    push_back( p_my_list, 3.9 );
    push_back( p_my_list, 8.1 );
    push_back( p_my_list, 2.6 );
}
```

- This stores:

```
4.7, 3.9, 8.1, 2.6
```


Going out of scope…

- Or another function that collects data:

```
void function_name() {
    Node *p_my_list{nullptr};

    while ( true ) {
       std::cout << "Enter a number (0 to finish): ";
       double x;
       std::cin >> x;

       if ( x == 0.0 ) {
           break;
       }

       push_back( p_my_list, x );
    }

    // Do something with the data
}
```

Imagine reading a sensor…

Going out of scope…

- In both cases, the memory for the allocated nodes is not freed
  - The local variable `p_my_list` goes out of scope, and its information is lost forever
  - We must have a mechanism for removing and freeing nodes from a linked list

Freeing all nodes

- Suppose we want to delete all nodes:

```
void freeing_all( Node *&p_list_head ) {
    while ( !is_empty( p_list_head ) ) {
        Node *p_current_head{ p_list_head };
        p_list_head = p_list_head->p_next_node_;
        delete p_current_head;
        p_current_head = nullptr;
    }
}
```

Freeing all nodes

- Or another function that collects data:

```
void function_name() {
    Node *p_my_list{nullptr};

    while ( true ) {
       std::cout << "Enter a number (0 to finish): ";
       double x;
       std::cin >> x;

       if ( x == 0.0 ) {
           break;
       }

       push_back( p_my_list, x );
    }

    // Do something with the data

    freeing_all( p_my_list );
    assert( p_my_list == nullptr );
}
```


Problems with this approach…

- What happens if the programmer authors code like the following:

```
if ( p_list_head = nullptr ) {
    // The list is empty...
} else {
    // Do something with the list...
}
```

Assigning `p_list_head` the value `nullptr`

or

```
if ( !is_empty( p_list_head ) ) {
    double value{ p_list_head->value_ };
    p_list_head = p_list_head->p_next_node_;

    // Do something with 'value'
}
```

Memory leak

Problems with this approach…

- Alternatively, suppose the programmer forgets to call `freeing_all(…)`

  Memory leak

- Alternatively, what happens with the following code?

```
int main() {
    Node *p_my_list;

    while ( sensor_ready() ) {
        push_back( p_my_list, sensor_get_data() );
    }

    freeing_all( p_my_data );

    return 0;
}
```

Wild pointer!

Problems with this approach…

- Allowing any programmer to access and manipulate the member variables of this linked list will invariably result in errors
- Goal:
  - To allow an experienced programmer to author a data structure that others can use
  - To prevent honest programmers from accidentally making mistakes in programming in using the data structure
- For this to happen:
  - The following operations should be automated:
    - The initialization of the data structure
    - Cleaning up the data structure
  - The programmer should not be able to assign to member variables

Summary

- Following this lesson, you now
  - Understand the design of a linked list with pointers
  - Know that this:
    - Requires allocation of memory
    - Requires the freeing of memory
  - Understand that access to member variables will cause problems if programmers make mistakes

References

[1] No references?

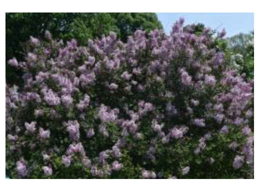
Colophon

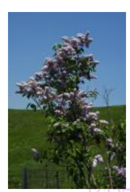
These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

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