Outline

- In this lesson, we will:
  - Explain why the `size()` function is slow
  - Show how we can speed this up with a `size_` member variable
  - Step through all member functions that must be modified to accommodate this variable:
    - The constructor
    - `size()`
    - `push_front(…)`
    - `pop_front()`
  - Determine the cost of adding this member variable

Our linked list class

- Reviewing our *linked list* class

```
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 value ) 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
};
```


Problem

- To get the size, we must count all the entries in the linked list
  - This could be hundreds, thousands or more
- Could this not potentially slow down an application?
  - Can we speed up the run-time of `size()`?

Problem

- One solution is to add a `size_` member variable:

```
class Node;
class Linked_list;

class Linked_list {
    public:
        // ...declarations of public member functions...

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


List size member variable

- Now:
  - The `size_` member variable can be immediately returned by the `size()` member function
  - We must initialize `size_` inside the constructor
  - We must update `size_` whenever we:
    - Push a new node
    - Pop a node from the linked list

Constructor and destructor

- The constructor must initialize the list size:

```
Linked_list::Linked_list():
p_list_head_{nullptr},
size_{0} {
    // Nothing else for the constructor to do
}
```

- In C++, you should initialize all member variables before the body of the constructor executes

Getting the size

- Now the `size()` function can simply return that variable:

```
void Linked_list::size() const {
    return size_;
}
```

Inserting a node

- When pushing a new node at the front of the linked list, we must increment the list size:

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


Removing a node

- When popping a node, we must decrement the list size

```
bool Linked_list::pop_front() {
    if ( empty() ) {
        return false;
    } else {
        assert( size() >= 1 );

        Node *p_current_head{ p_list_head_ };
        p_list_head_ = p_list_head_->p_next_node_;

        delete p_current_head;
        p_current_head = nullptr;

        --size_;
        return true;
    }
}
```


Clearing all nodes in a list

- Why do we not have to update `clear()`?

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


Clearing all nodes in a list

- We can speed up `clear()`?

```
double Linked_list::operator[]( std::size_t const n ) const {
  if ( size() <= n ) {
    return 0.0;
  } else {
  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_;
  }

  assert( false );    // We should never get here
  return 0.0;
 }
}
```

Our linked list class

- Our public member functions have not changed their behavior:
  - The *interface* has not changed

```
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 value ) 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
        std::size_t size_;
};
```


Benefit of encapsulation

- Adding this member variable and modifying our member functions in no way affected the way users interact with this class
- The only differences are:
  - It uses a little more memory (4 to 8 bytes)
  - Some functions are trivially slower (one extra instruction)
  - The `size()` function is significantly faster now

Summary

- Following this lesson, you now
  - Understand we can modify a class without affecting the user
  - Know how a list size member variable can significantly decrease the execution time of the `size()` function
  - Understand that the cost is a small increase in memory and run time

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.