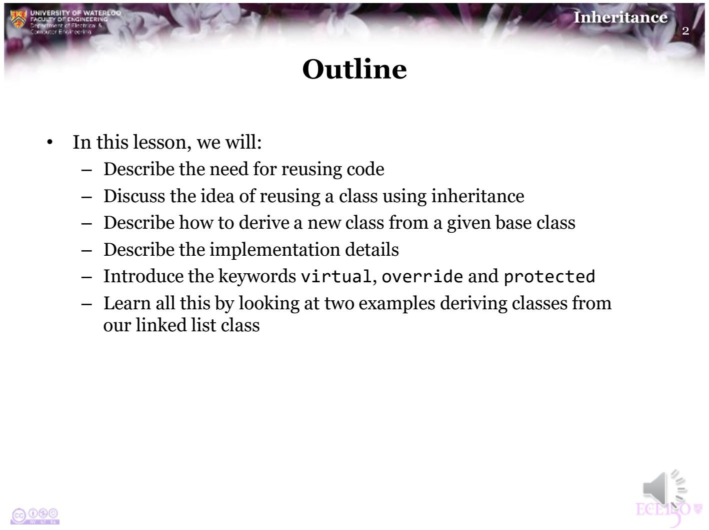
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

Inheritance

Douglas Wilhelm Harder, M.Math. LEL
Prof. Hiren Patel, Ph.D., P.Eng.
Prof. Werner Dietl, Ph.D.

1

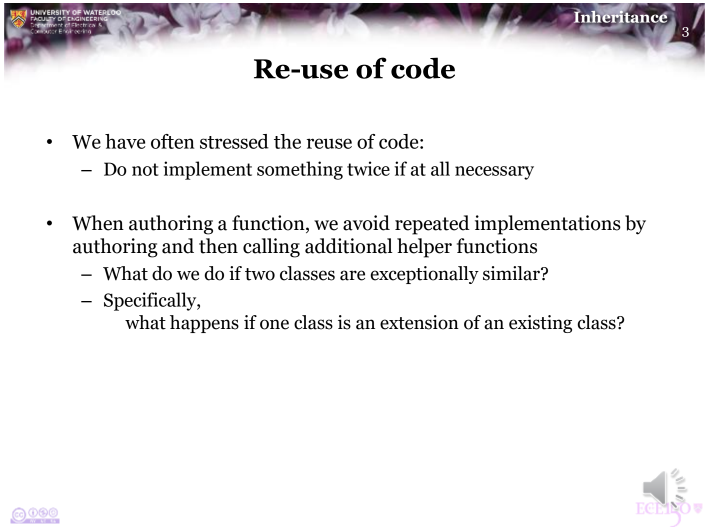


Inheritance 2

Outline

- In this lesson, we will:
  - Describe the need for reusing code
  - Discuss the idea of reusing a class using inheritance
  - Describe how to derive a new class from a given base class
  - Describe the implementation details
  - Introduce the keywords `virtual`, `override` and `protected`
  - Learn all this by looking at two examples deriving classes from our linked list class

2



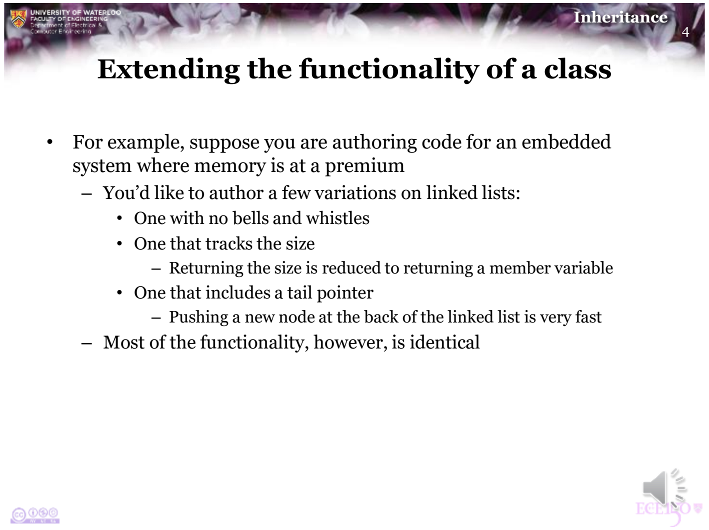
Inheritance 3

Re-use of code

- We have often stressed the reuse of code:
  - Do not implement something twice if at all necessary

- When authoring a function, we avoid repeated implementations by authoring and then calling additional helper functions
  - What do we do if two classes are exceptionally similar?
  - Specifically,
    what happens if one class is an extension of an existing class?

3

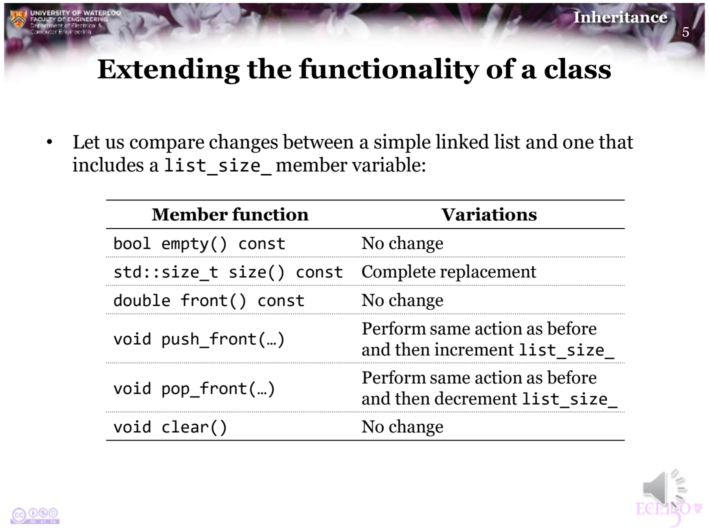


Inheritance 4

Extending the functionality of a class

- For example, suppose you are authoring code for an embedded system where memory is at a premium
  - You'd like to author a few variations on linked lists:
    - One with no bells and whistles
    - One that tracks the size
      - Returning the size is reduced to returning a member variable
    - One that includes a tail pointer
      - Pushing a new node at the back of the linked list is very fast
  - Most of the functionality, however, is identical

4

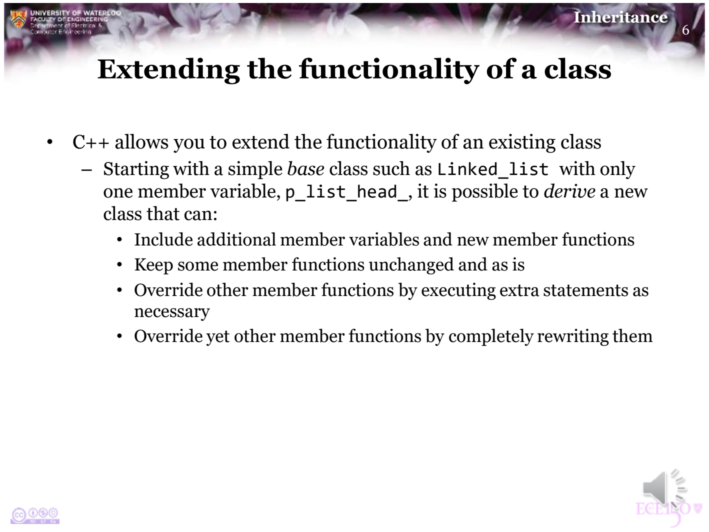


Extending the functionality of a class

- Let us compare changes between a simple linked list and one that includes a `list_size_` member variable:

Member function	Variations
`bool empty() const`	No change
`std::size_t size() const`	Complete replacement
`double front() const`	No change
`void push_front(…)`	Perform same action as before and then increment `list_size_`
`void pop_front(…)`	Perform same action as before and then decrement `list_size_`
`void clear()`	No change

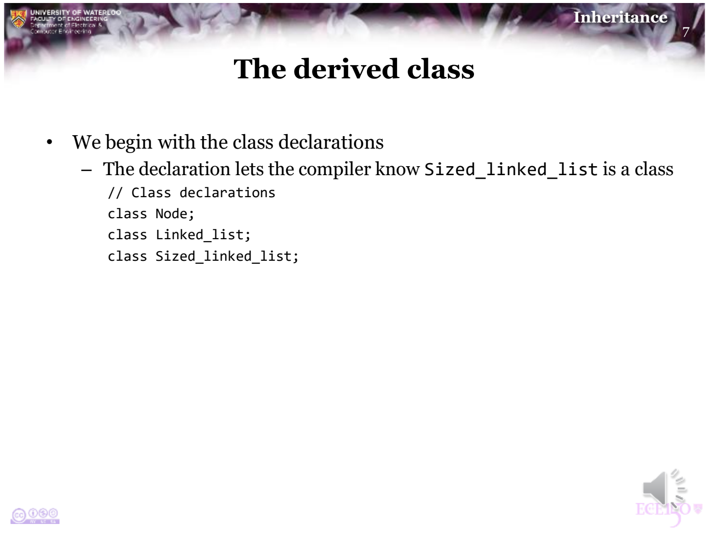
5



Extending the functionality of a class

- C++ allows you to extend the functionality of an existing class
  - Starting with a simple *base* class such as `Linked_list` with only one member variable, `p_list_head_`, it is possible to *derive* a new class that can:
    - Include additional member variables and new member functions
    - Keep some member functions unchanged and as is
    - Override other member functions by executing extra statements as necessary
    - Override yet other member functions by completely rewriting them

6

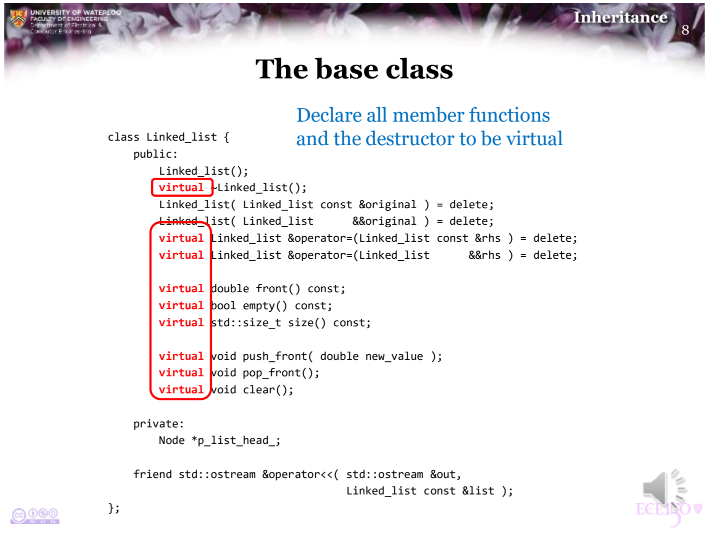


The derived class

- We begin with the class declarations
  - The declaration lets the compiler know `Sized_linked_list` is a class

```
// Class declarations
class Node;
class Linked_list;
class Sized_linked_list;
```

7

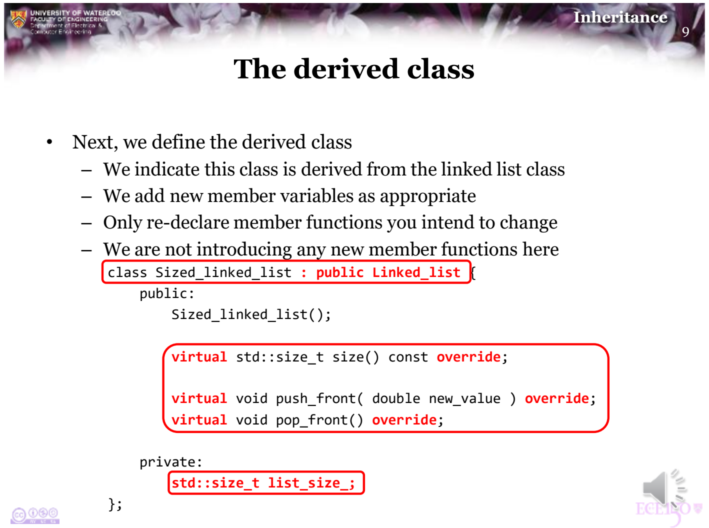


The base class

Declare all member functions and the destructor to be virtual

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

        virtual double front() const;
        virtual bool empty() const;
        virtual std::size_t size() const;

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

    private:
        Node *p_list_head_;

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

8

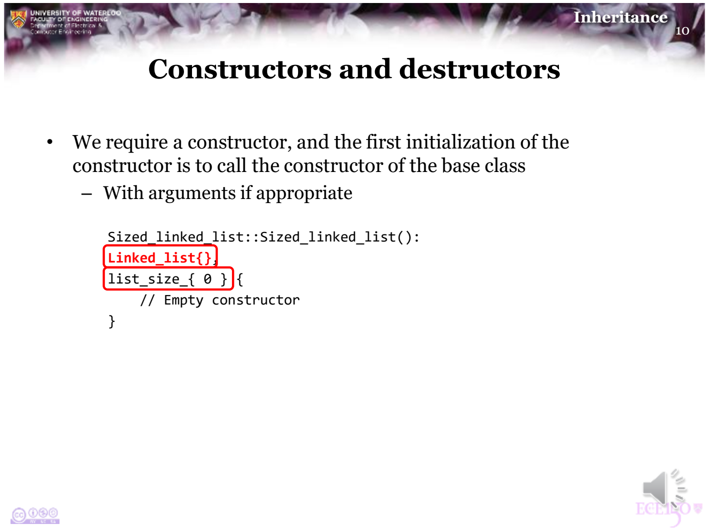


The derived class

- Next, we define the derived class
  - We indicate this class is derived from the linked list class
  - We add new member variables as appropriate
  - Only re-declare member functions you intend to change
  - We are not introducing any new member functions here

```
class Sized_linked_list : public Linked_list {
    public:
        Sized_linked_list();

        virtual std::size_t size() const override;

        virtual void push_front( double new_value ) override;
        virtual void pop_front() override;

    private:
        std::size_t list_size_;
};
```

9

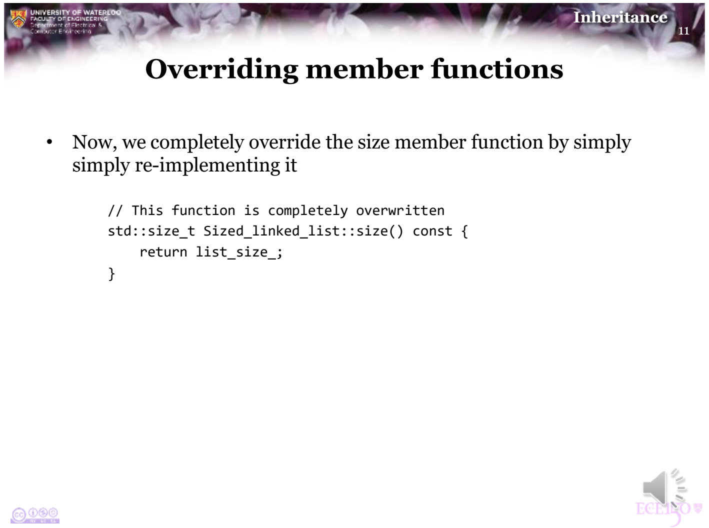


Constructors and destructors

- We require a constructor, and the first initialization of the constructor is to call the constructor of the base class
  - With arguments if appropriate

```
Sized_linked_list::Sized_linked_list():
Linked_list{},
list_size_{ 0 } {
    // Empty constructor
}
```

10

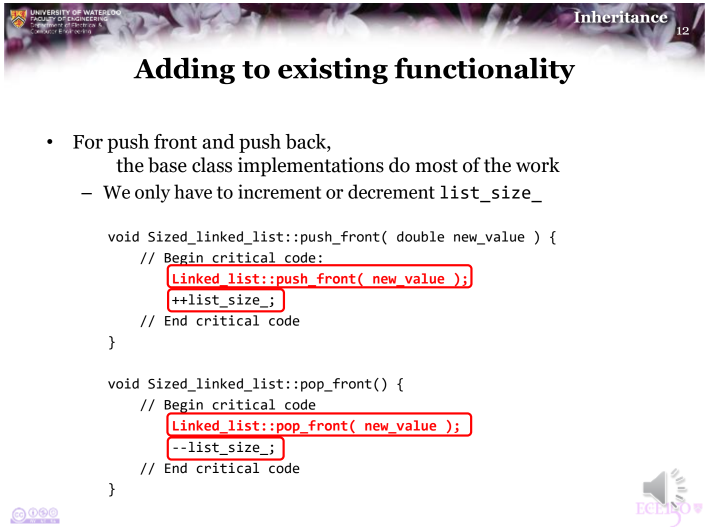


Overriding member functions

- Now, we completely override the size member function by simply simply re-implementing it

```
// This function is completely overwritten
std::size_t Sized_linked_list::size() const {
    return list_size_;
}
```

11

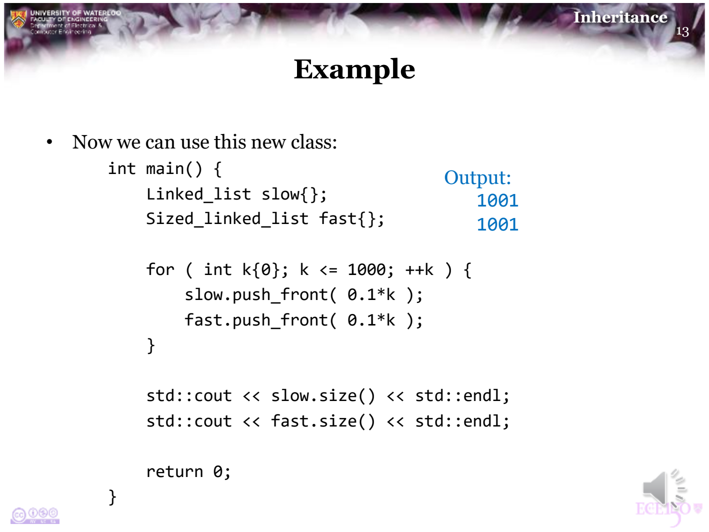


Adding to existing functionality

- For push front and push back,
  the base class implementations do most of the work
  - We only have to increment or decrement `list_size_`

```
void Sized_linked_list::push_front( double new_value ) {
    // Begin critical code:
        Linked_list::push_front( new_value );
        ++list_size_;
    // End critical code
}

void Sized_linked_list::pop_front() {
    // Begin critical code
        Linked_list::pop_front( new_value );
        --list_size_;
    // End critical code
}
```

12



Example

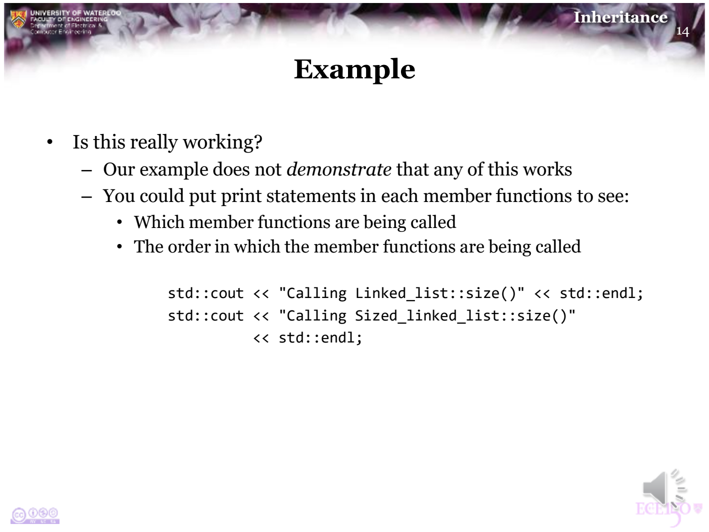
- Now we can use this new class:

```
int main() {
    Linked_list slow{};
    Sized_linked_list fast{};

    for ( int k{0}; k <= 1000; ++k ) {
        slow.push_front( 0.1*k );
        fast.push_front( 0.1*k );
    }

    std::cout << slow.size() << std::endl;
    std::cout << fast.size() << std::endl;

    return 0;
}
```

Output:

```
1001
1001
```

13

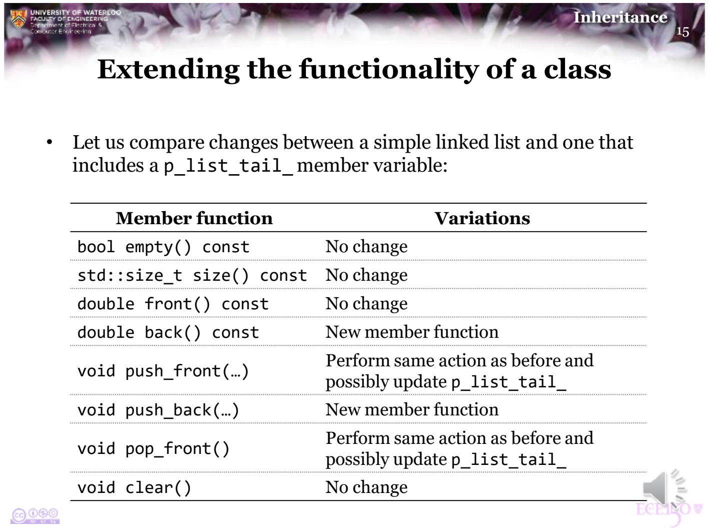


Example

- Is this really working?
  - Our example does not *demonstrate* that any of this works
  - You could put print statements in each member functions to see:
    - Which member functions are being called
    - The order in which the member functions are being called

```
std::cout << "Calling Linked_list::size()" << std::endl;
std::cout << "Calling Sized_linked_list::size()"
          << std::endl;
```

14

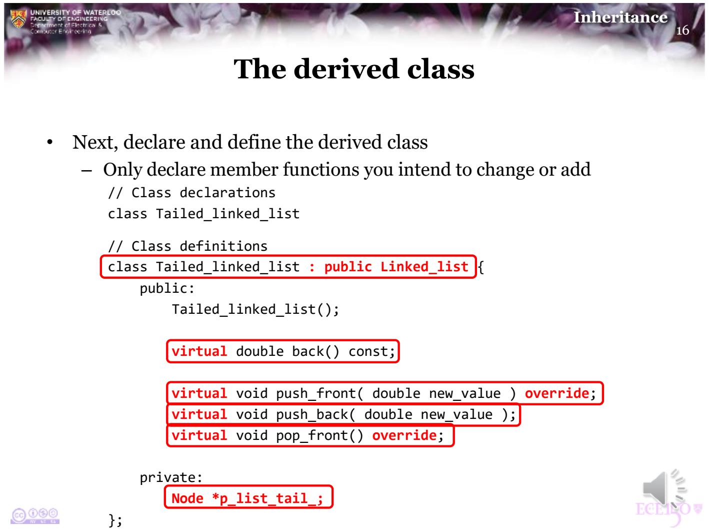


Extending the functionality of a class

- Let us compare changes between a simple linked list and one that includes a p_list_tail_ member variable:

Member function	Variations
bool empty() const	No change
std::size_t size() const	No change
double front() const	No change
double back() const	New member function
void push_front(…)	Perform same action as before and possibly update p_list_tail_
void push_back(…)	New member function
void pop_front()	Perform same action as before and possibly update p_list_tail_
void clear()	No change

15

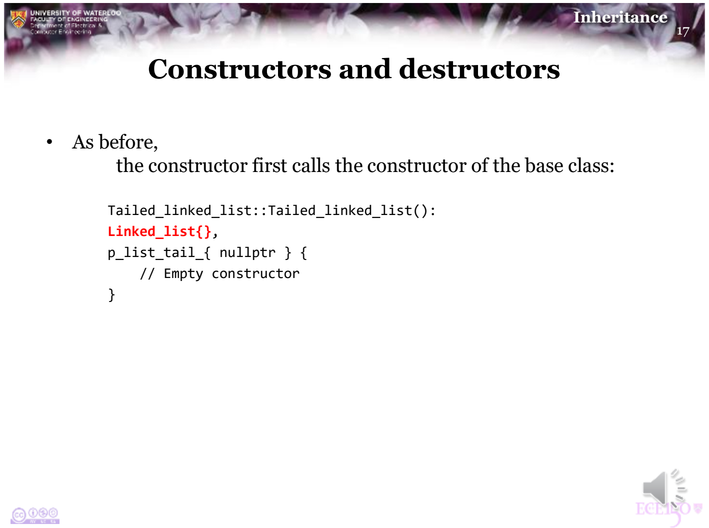


The derived class

- Next, declare and define the derived class
  - Only declare member functions you intend to change or add

```
// Class declarations
class Tailed_linked_list

// Class definitions
class Tailed_linked_list : public Linked_list{
    public:
        Tailed_linked_list();

        virtual double back() const;

        virtual void push_front( double new_value ) override;
        virtual void push_back( double new_value );
        virtual void pop_front() override;

    private:
        Node *p_list_tail_;
};
```

16

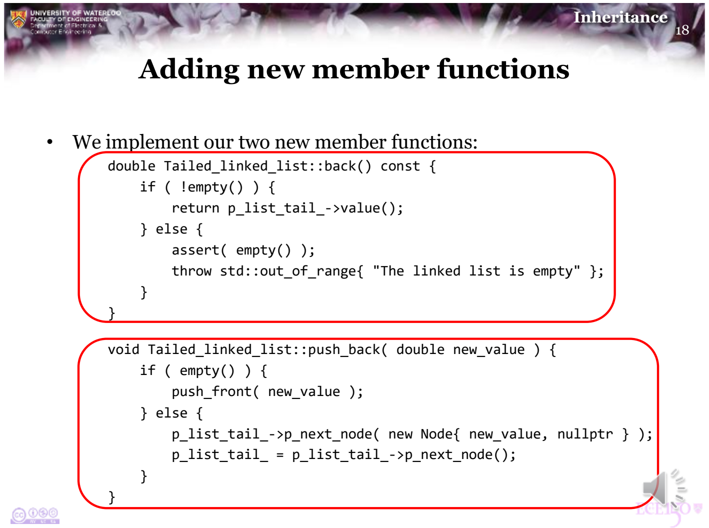


Constructors and destructors

- As before,
  the constructor first calls the constructor of the base class:

```
Tailed_linked_list::Tailed_linked_list():
Linked_list{},
p_list_tail_{ nullptr } {
    // Empty constructor
}
```

17



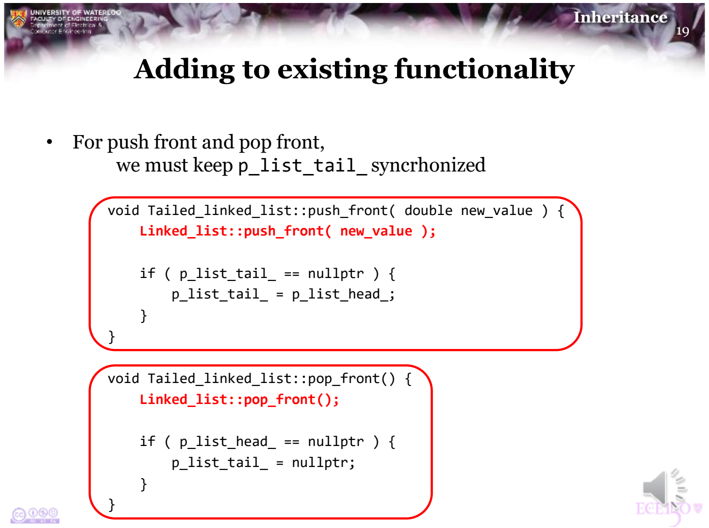
Adding new member functions

- We implement our two new member functions:

```
double Tailed_linked_list::back() const {
    if ( !empty() ) {
        return p_list_tail_->value();
    } else {
        assert( empty() );
        throw std::out_of_range{ "The linked list is empty" };
    }
}
```

```
void Tailed_linked_list::push_back( double new_value ) {
    if ( empty() ) {
        push_front( new_value );
    } else {
        p_list_tail_->p_next_node( new Node{ new_value, nullptr } );
        p_list_tail_ = p_list_tail_->p_next_node();
    }
}
```

18



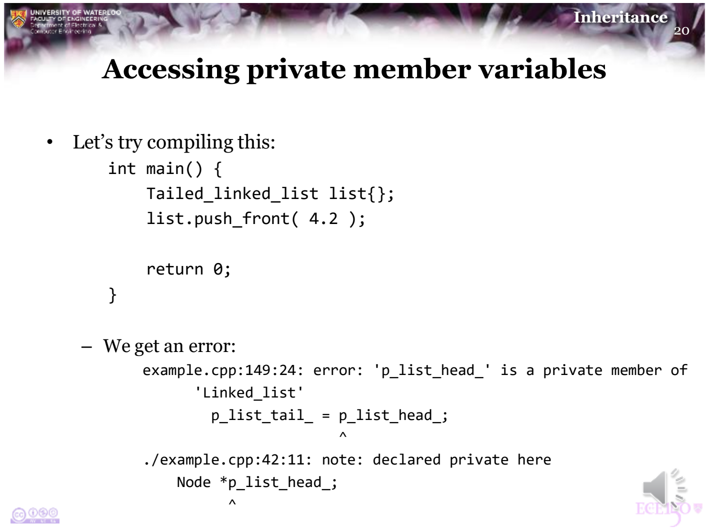
Adding to existing functionality

- For push front and pop front,
  we must keep `p_list_tail_` syncrhonized

```
void Tailed_linked_list::push_front( double new_value ) {
    Linked_list::push_front( new_value );

    if ( p_list_tail_ == nullptr ) {
        p_list_tail_ = p_list_head_;
    }
}
```

```
void Tailed_linked_list::pop_front() {
    Linked_list::pop_front();

    if ( p_list_head_ == nullptr ) {
        p_list_tail_ = nullptr;
    }
}
```

19



Accessing private member variables

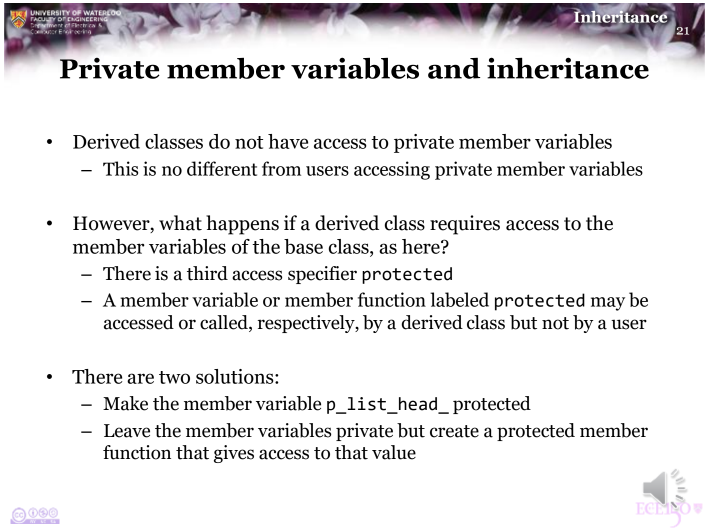
- Let's try compiling this:

```
int main() {
    Tailed_linked_list list{};
    list.push_front( 4.2 );

    return 0;
}
```

  – We get an error:

```
example.cpp:149:24: error: 'p_list_head_' is a private member of
        'Linked_list'
          p_list_tail_ = p_list_head_;
                         ^
./example.cpp:42:11: note: declared private here
    Node *p_list_head_;
          ^
```

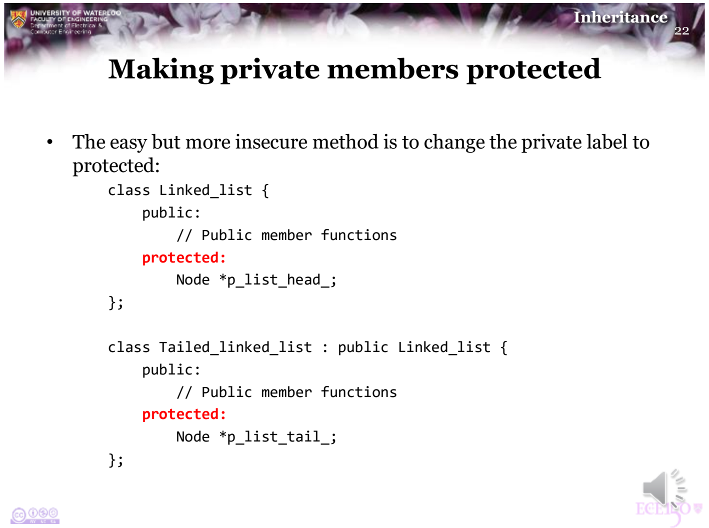
20



Private member variables and inheritance

- Derived classes do not have access to private member variables
  - This is no different from users accessing private member variables
- However, what happens if a derived class requires access to the member variables of the base class, as here?
  - There is a third access specifier `protected`
  - A member variable or member function labeled `protected` may be accessed or called, respectively, by a derived class but not by a user
- There are two solutions:
  - Make the member variable `p_list_head_` protected
  - Leave the member variables private but create a protected member function that gives access to that value

21

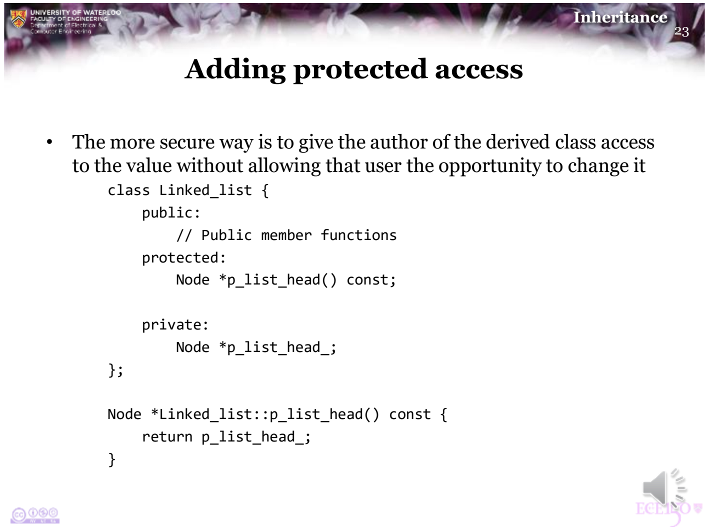


Making private members protected

- The easy but more insecure method is to change the private label to protected:

```
class Linked_list {
    public:
        // Public member functions
    protected:
        Node *p_list_head_;
};

class Tailed_linked_list : public Linked_list {
    public:
        // Public member functions
    protected:
        Node *p_list_tail_;
};
```

22

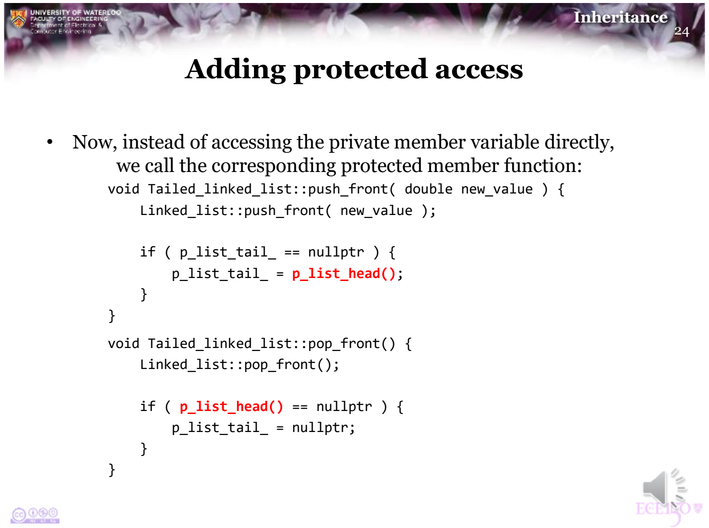


Adding protected access

- The more secure way is to give the author of the derived class access to the value without allowing that user the opportunity to change it

```
class Linked_list {
    public:
        // Public member functions
    protected:
        Node *p_list_head() const;

    private:
        Node *p_list_head_;
};

Node *Linked_list::p_list_head() const {
    return p_list_head_;
}
```

23

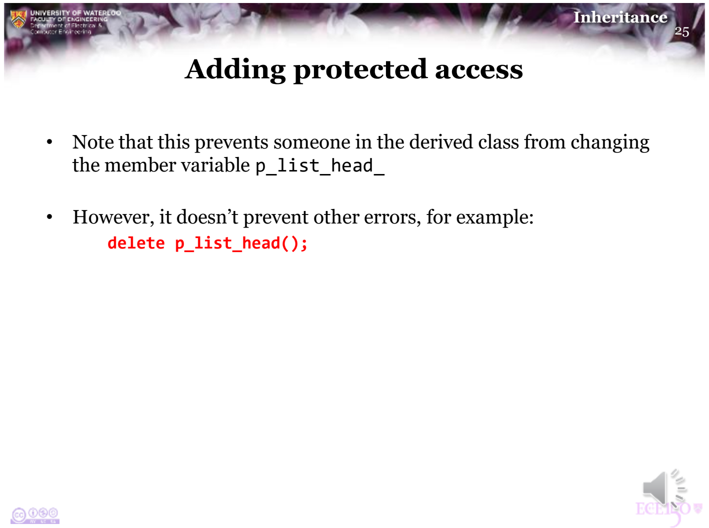


Adding protected access

- Now, instead of accessing the private member variable directly, we call the corresponding protected member function:

```
void Tailed_linked_list::push_front( double new_value ) {
    Linked_list::push_front( new_value );

    if ( p_list_tail_ == nullptr ) {
        p_list_tail_ = p_list_head();
    }
}

void Tailed_linked_list::pop_front() {
    Linked_list::pop_front();

    if ( p_list_head() == nullptr ) {
        p_list_tail_ = nullptr;
    }
}
```

24

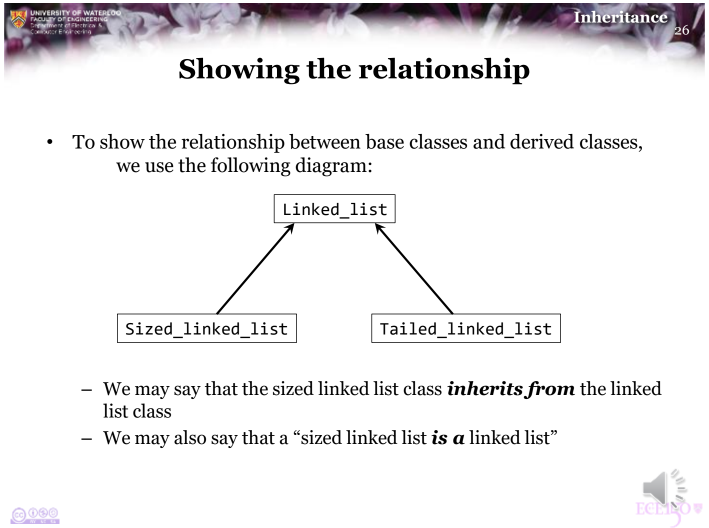


Adding protected access

- Note that this prevents someone in the derived class from changing the member variable `p_list_head_`
- However, it doesn't prevent other errors, for example:

```
delete p_list_head();
```

25



Showing the relationship

- To show the relationship between base classes and derived classes, we use the following diagram:

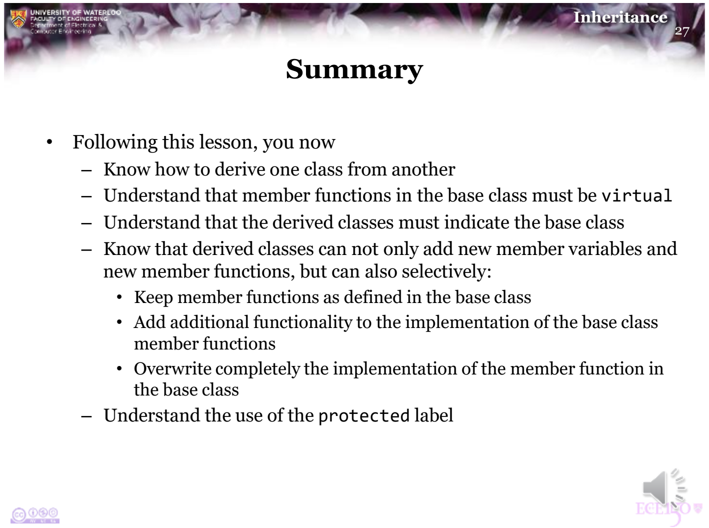
Linked_list

Sized_linked_list

Tailed_linked_list

  - We may say that the sized linked list class ***inherits from*** the linked list class
  - We may also say that a "sized linked list ***is a*** linked list"

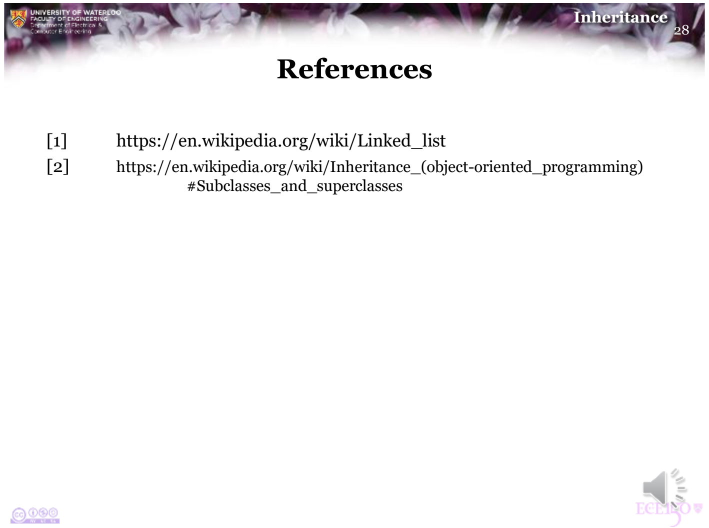
26



Summary

- Following this lesson, you now
  - Know how to derive one class from another
  - Understand that member functions in the base class must be `virtual`
  - Understand that the derived classes must indicate the base class
  - Know that derived classes can not only add new member variables and new member functions, but can also selectively:
    - Keep member functions as defined in the base class
    - Add additional functionality to the implementation of the base class member functions
    - Overwrite completely the implementation of the member function in the base class
  - Understand the use of the `protected` label

27

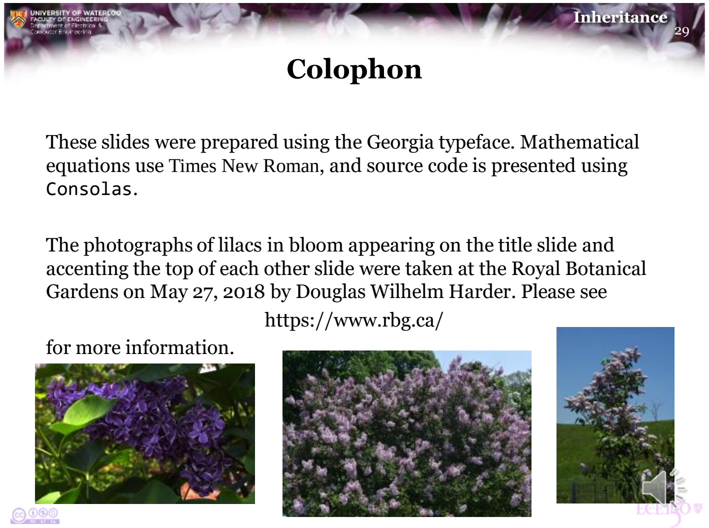


References

[1] https://en.wikipedia.org/wiki/Linked_list

[2] https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)#Subclasses_and_superclasses

28



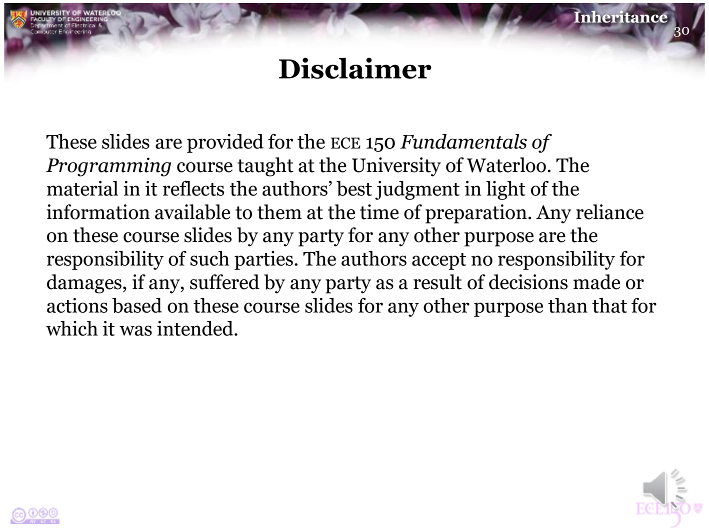
Colophon

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.