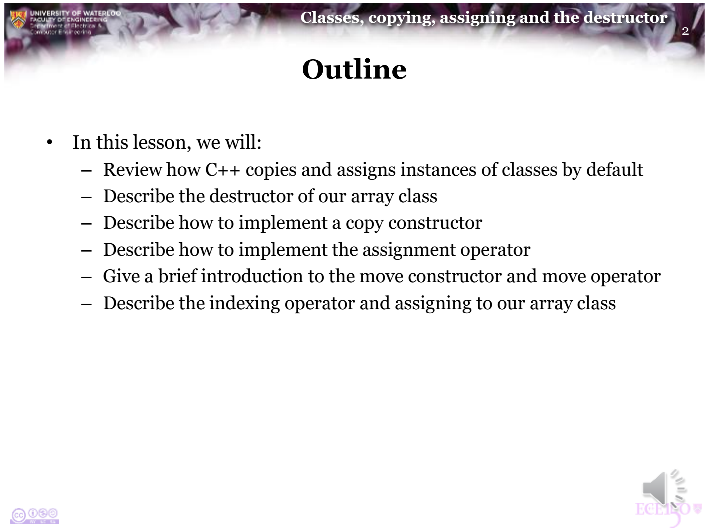
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

Classes, copies, assignment and the destructor

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

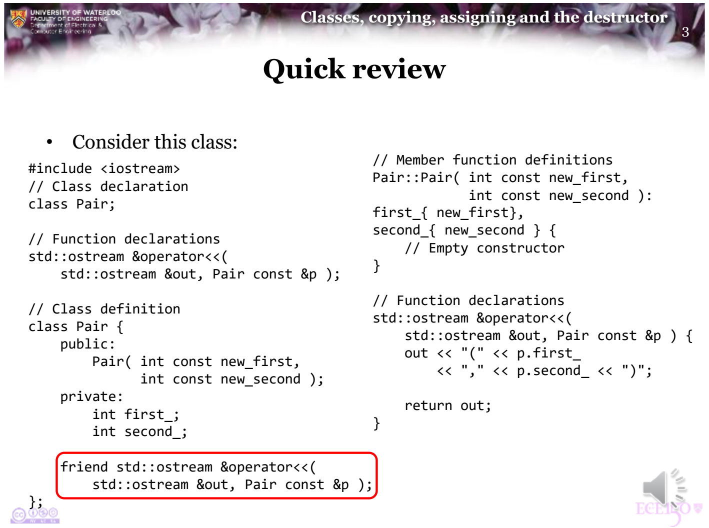
1



Outline

- In this lesson, we will:
  - Review how C++ copies and assigns instances of classes by default
  - Describe the destructor of our array class
  - Describe how to implement a copy constructor
  - Describe how to implement the assignment operator
  - Give a brief introduction to the move constructor and move operator
  - Describe the indexing operator and assigning to our array class

2



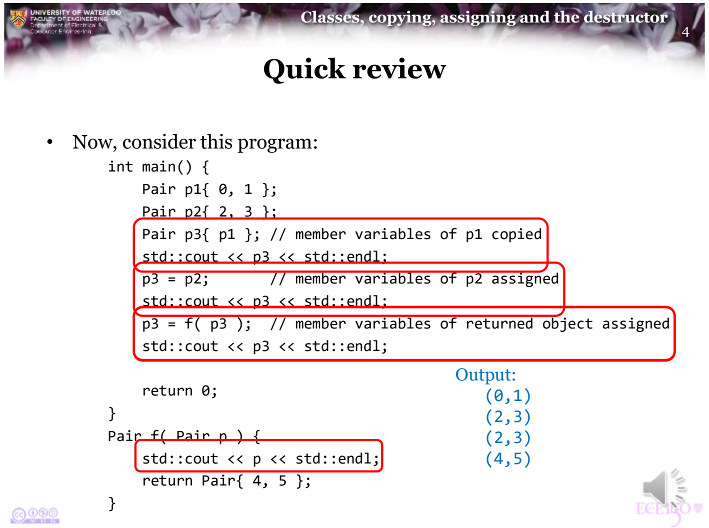
Quick review

- Consider this class:

```
#include <iostream>
// Class declaration
class Pair;

// Function declarations
std::ostream &operator<<(
    std::ostream &out, Pair const &p );

// Class definition
class Pair {
    public:
        Pair( int const new_first,
              int const new_second );
    private:
        int first_;
        int second_;

    friend std::ostream &operator<<(
        std::ostream &out, Pair const &p );
};
```

```
// Member function definitions
Pair::Pair( int const new_first,
            int const new_second ):
first_{ new_first},
second_{ new_second } {
    // Empty constructor
}

// Function declarations
std::ostream &operator<<(
    std::ostream &out, Pair const &p ) {
    out << "(" << p.first_
        << "," << p.second_ << ")";

    return out;
}
```

3



Quick review

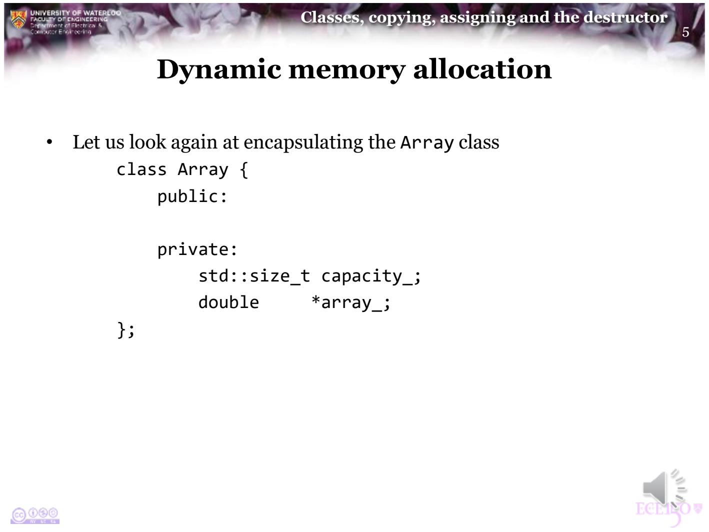
- Now, consider this program:

```
int main() {
    Pair p1{ 0, 1 };
    Pair p2{ 2, 3 };
    Pair p3{ p1 }; // member variables of p1 copied
    std::cout << p3 << std::endl;
    p3 = p2;       // member variables of p2 assigned
    std::cout << p3 << std::endl;
    p3 = f( p3 );  // member variables of returned object assigned
    std::cout << p3 << std::endl;

    return 0;
}
Pair f( Pair p ) {
    std::cout << p << std::endl;
    return Pair{ 4, 5 };
}
```

Output:

```
(0,1)
(2,3)
(2,3)
(4,5)
```

4

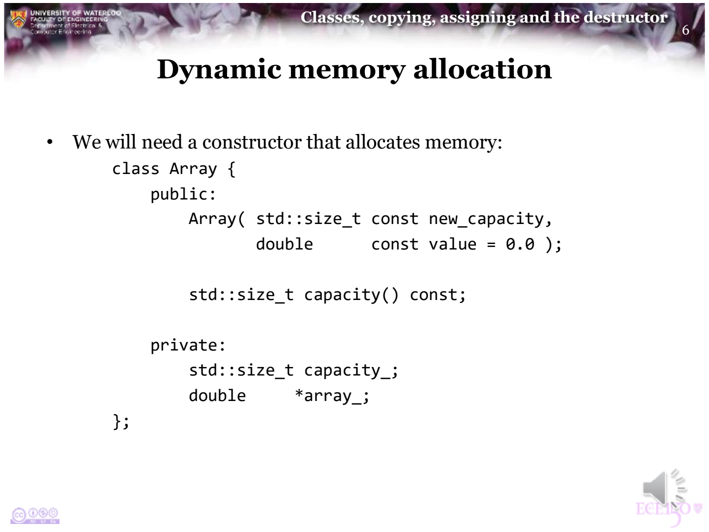


Dynamic memory allocation

- Let us look again at encapsulating the `Array` class

```
class Array {
    public:

    private:
        std::size_t capacity_;
        double     *array_;
};
```

5

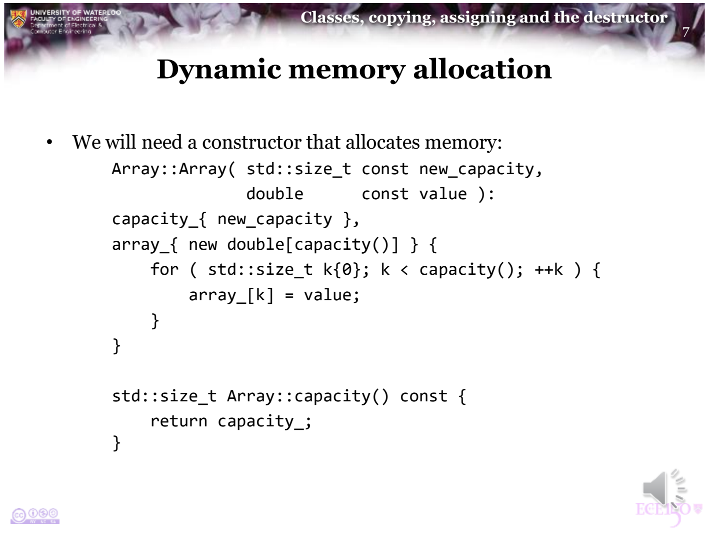


Dynamic memory allocation

- We will need a constructor that allocates memory:

```
class Array {
    public:
        Array( std::size_t const new_capacity,
               double      const value = 0.0 );

        std::size_t capacity() const;

    private:
        std::size_t capacity_;
        double     *array_;
};
```

6

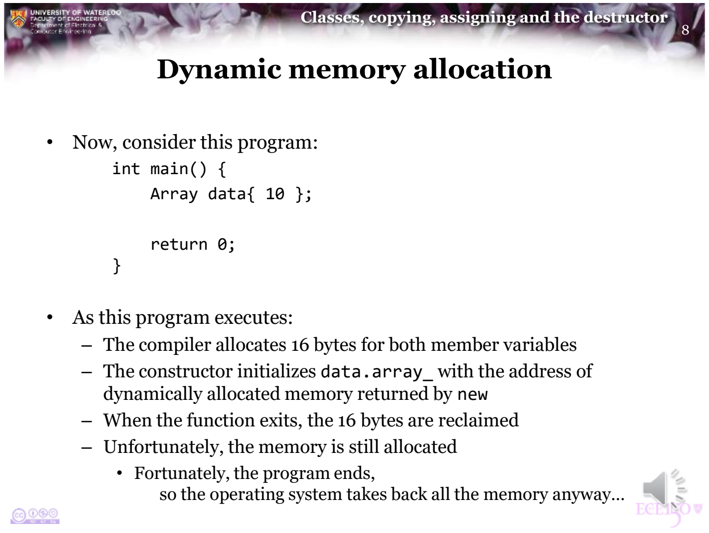


Dynamic memory allocation

- We will need a constructor that allocates memory:

```
Array::Array( std::size_t const new_capacity,
              double      const value ):
capacity_{ new_capacity },
array_{ new double[capacity()] } {
    for ( std::size_t k{0}; k < capacity(); ++k ) {
        array_[k] = value;
    }
}

std::size_t Array::capacity() const {
    return capacity_;
}
```

7



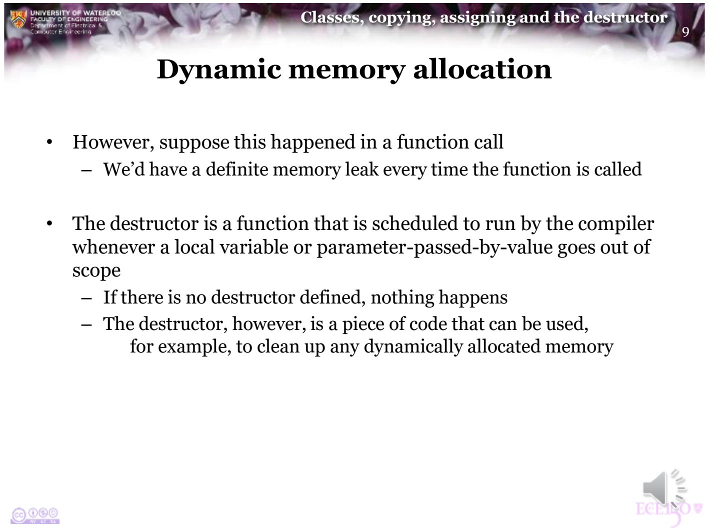
Dynamic memory allocation

- Now, consider this program:

```
int main() {
    Array data{ 10 };

    return 0;
}
```

- As this program executes:
  - The compiler allocates 16 bytes for both member variables
  - The constructor initializes `data.array_` with the address of dynamically allocated memory returned by `new`
  - When the function exits, the 16 bytes are reclaimed
  - Unfortunately, the memory is still allocated
    - Fortunately, the program ends,
      so the operating system takes back all the memory anyway…

8

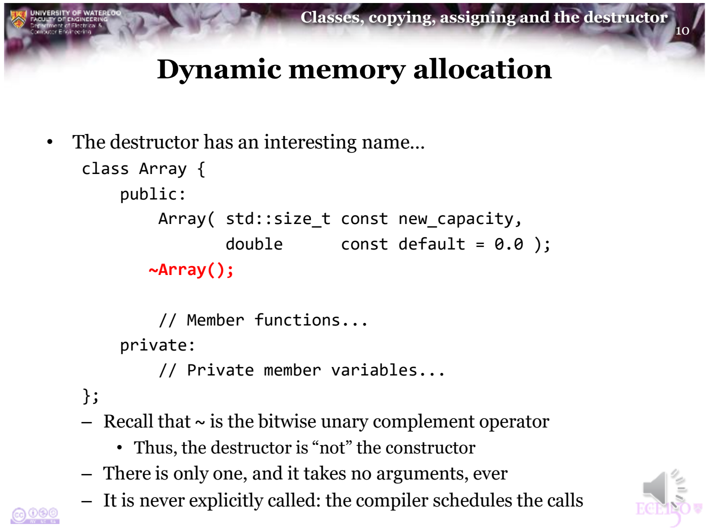


Classes, copying, assigning and the destructor 9

Dynamic memory allocation

- However, suppose this happened in a function call
  - We'd have a definite memory leak every time the function is called
- The destructor is a function that is scheduled to run by the compiler whenever a local variable or parameter-passed-by-value goes out of scope
  - If there is no destructor defined, nothing happens
  - The destructor, however, is a piece of code that can be used, for example, to clean up any dynamically allocated memory

9



Classes, copying, assigning and the destructor 10

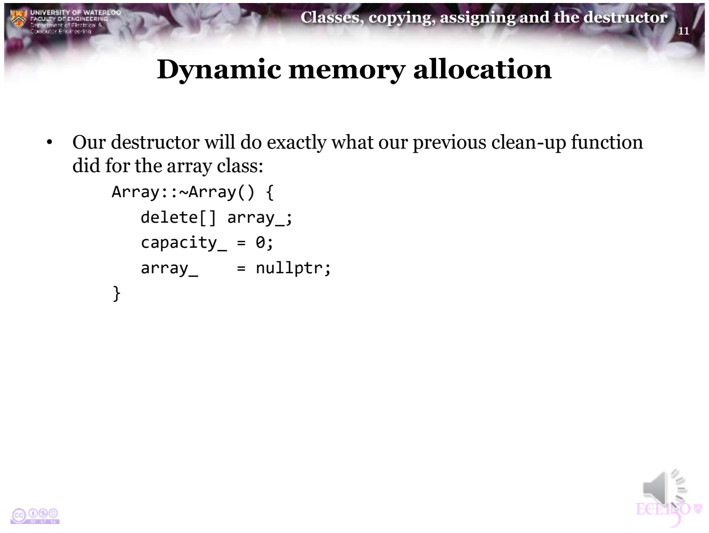
Dynamic memory allocation

- The destructor has an interesting name...

```
class Array {
    public:
        Array( std::size_t const new_capacity,
               double      const default = 0.0 );
       ~Array();

        // Member functions...
    private:
        // Private member variables...
};
```

  - Recall that ~ is the bitwise unary complement operator
    - Thus, the destructor is "not" the constructor
  - There is only one, and it takes no arguments, ever
  - It is never explicitly called: the compiler schedules the calls

10



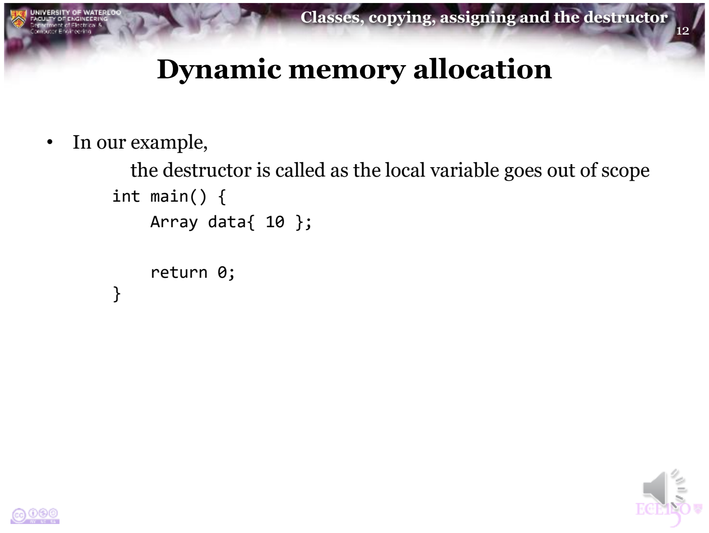
Classes, copying, assigning and the destructor 11

Dynamic memory allocation

- Our destructor will do exactly what our previous clean-up function did for the array class:

```
Array::~Array() {
   delete[] array_;
   capacity_ = 0;
   array_    = nullptr;
}
```

11



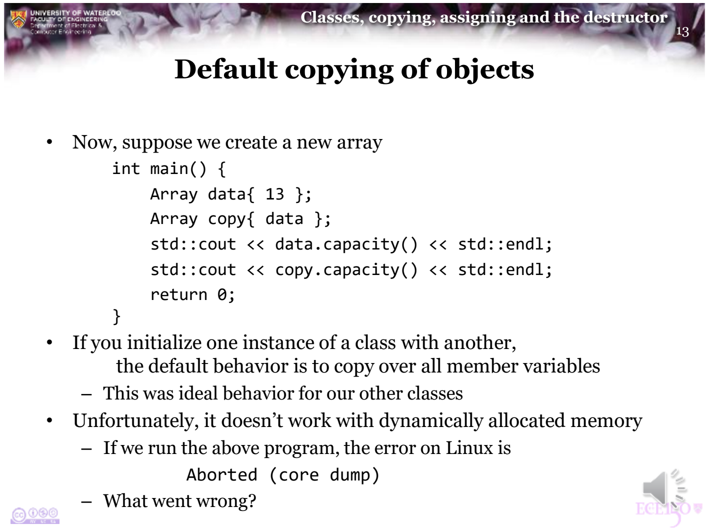
Classes, copying, assigning and the destructor 12

Dynamic memory allocation

- In our example,
  the destructor is called as the local variable goes out of scope

```
int main() {
    Array data{ 10 };

    return 0;
}
```

12



Default copying of objects

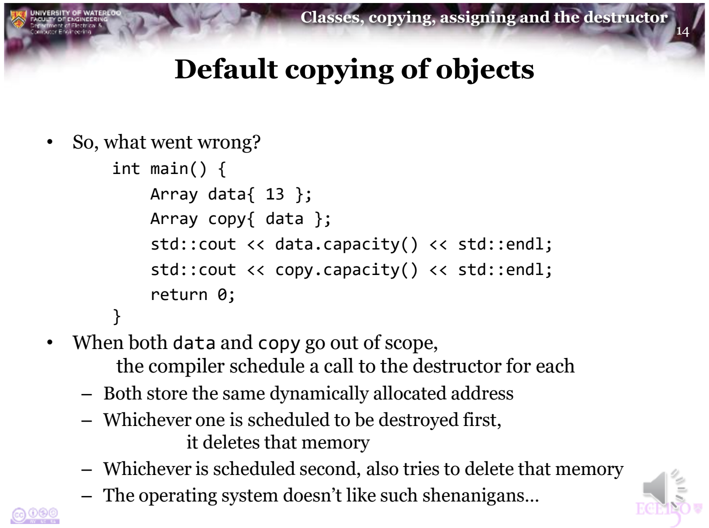
- Now, suppose we create a new array

```
int main() {
    Array data{ 13 };
    Array copy{ data };
    std::cout << data.capacity() << std::endl;
    std::cout << copy.capacity() << std::endl;
    return 0;
}
```

- If you initialize one instance of a class with another, the default behavior is to copy over all member variables
  - This was ideal behavior for our other classes
- Unfortunately, it doesn't work with dynamically allocated memory
  - If we run the above program, the error on Linux is

```
Aborted (core dump)
```

  - What went wrong?

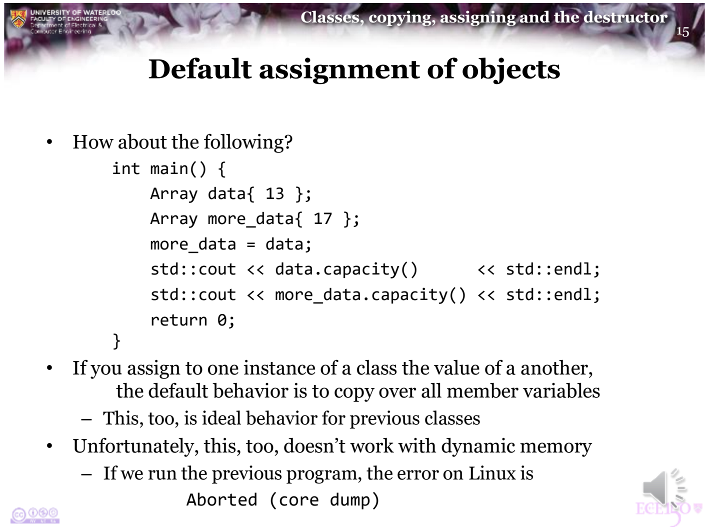


Default copying of objects

- So, what went wrong?

```
int main() {
    Array data{ 13 };
    Array copy{ data };
    std::cout << data.capacity() << std::endl;
    std::cout << copy.capacity() << std::endl;
    return 0;
}
```

- When both `data` and `copy` go out of scope, the compiler schedule a call to the destructor for each
  - Both store the same dynamically allocated address
  - Whichever one is scheduled to be destroyed first, it deletes that memory
  - Whichever is scheduled second, also tries to delete that memory
  - The operating system doesn't like such shenanigans…



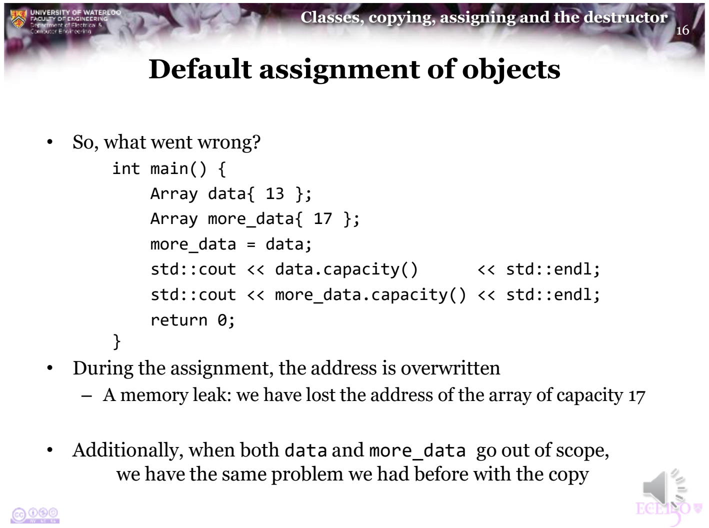
Default assignment of objects

- How about the following?

```
int main() {
    Array data{ 13 };
    Array more_data{ 17 };
    more_data = data;
    std::cout << data.capacity()      << std::endl;
    std::cout << more_data.capacity() << std::endl;
    return 0;
}
```

- If you assign to one instance of a class the value of a another, the default behavior is to copy over all member variables
  - This, too, is ideal behavior for previous classes
- Unfortunately, this, too, doesn't work with dynamic memory
  - If we run the previous program, the error on Linux is

```
Aborted (core dump)
```

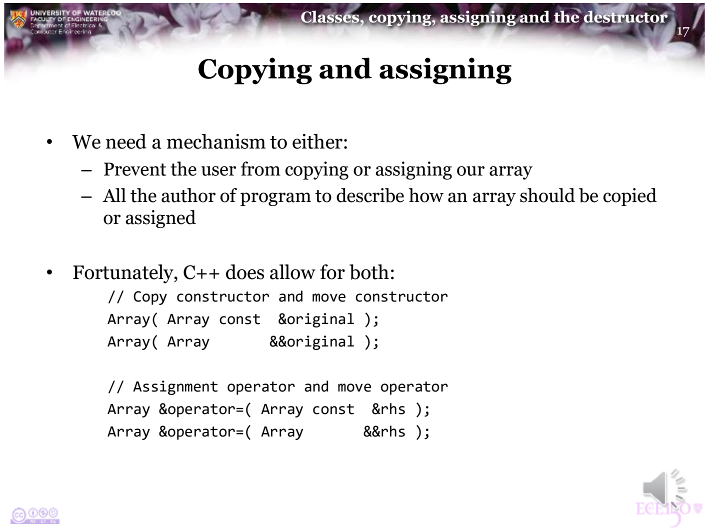


Default assignment of objects

- So, what went wrong?

```
int main() {
    Array data{ 13 };
    Array more_data{ 17 };
    more_data = data;
    std::cout << data.capacity()      << std::endl;
    std::cout << more_data.capacity() << std::endl;
    return 0;
}
```

- During the assignment, the address is overwritten
  - A memory leak: we have lost the address of the array of capacity 17
- Additionally, when both `data` and `more_data` go out of scope, we have the same problem we had before with the copy

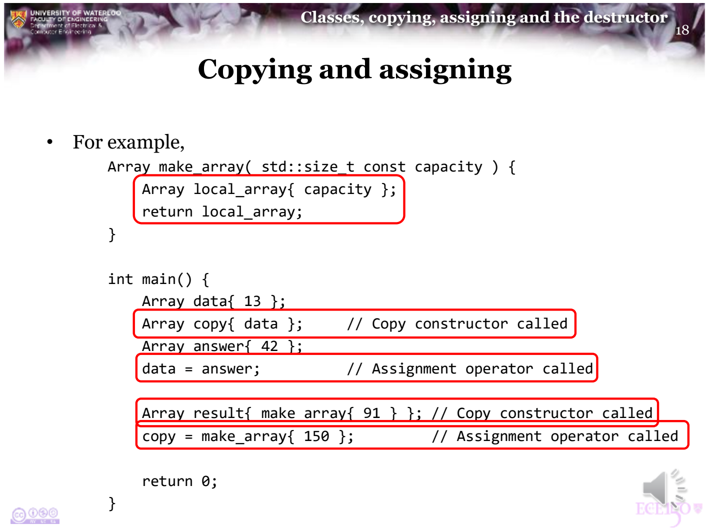


Copying and assigning

- We need a mechanism to either:
  - Prevent the user from copying or assigning our array
  - All the author of program to describe how an array should be copied or assigned
- Fortunately, C++ does allow for both:

```
// Copy constructor and move constructor
Array( Array const  &original );
Array( Array       &&original );

// Assignment operator and move operator
Array &operator=( Array const  &rhs );
Array &operator=( Array       &&rhs );
```

17

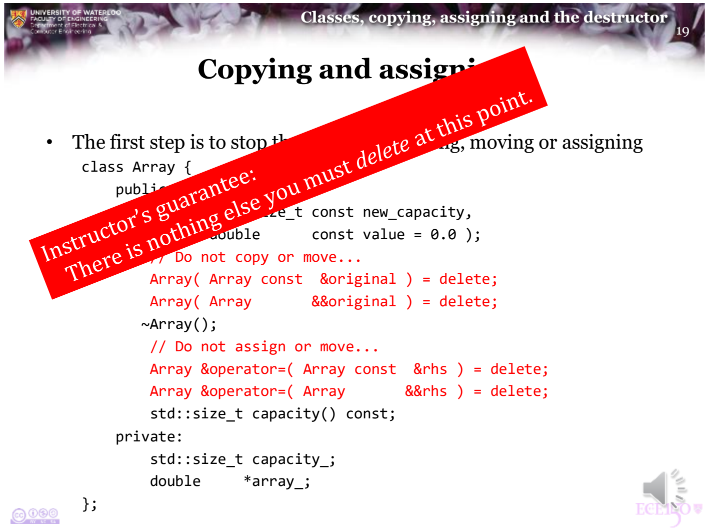


Copying and assigning

- For example,

```
Array make_array( std::size_t const capacity ) {
    Array local_array{ capacity };
    return local_array;
}

int main() {
    Array data{ 13 };
    Array copy{ data };      // Copy constructor called
    Array answer{ 42 };
    data = answer;           // Assignment operator called

    Array result{ make_array{ 91 } }; // Copy constructor called
    copy = make_array{ 150 };         // Assignment operator called

    return 0;
}
```

18



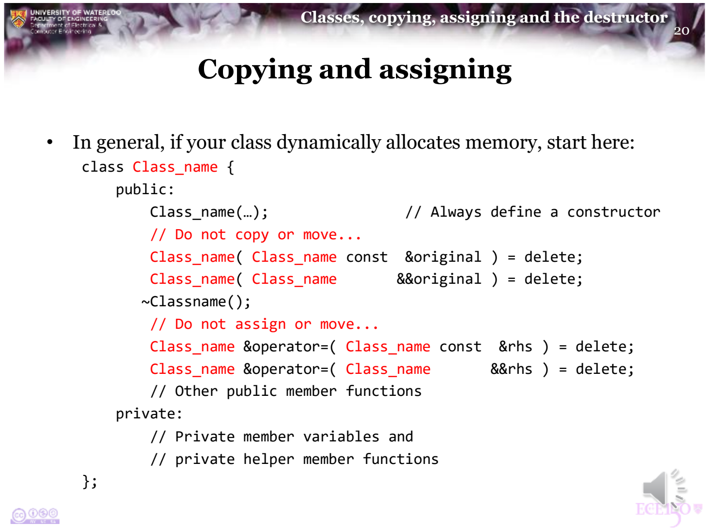
Copying and assigni

Instructor's guarantee:
There is nothing else you must *delete* at this point.

- The first step is to stop th ng, moving or assigning

```
class Array {
    publi
              ze_t const new_capacity,
              ouble      const value = 0.0 );
        // Do not copy or move...
        Array( Array const  &original ) = delete;
        Array( Array       &&original ) = delete;
       ~Array();
        // Do not assign or move...
        Array &operator=( Array const  &rhs ) = delete;
        Array &operator=( Array       &&rhs ) = delete;
        std::size_t capacity() const;
    private:
        std::size_t capacity_;
        double     *array_;
};
```

19

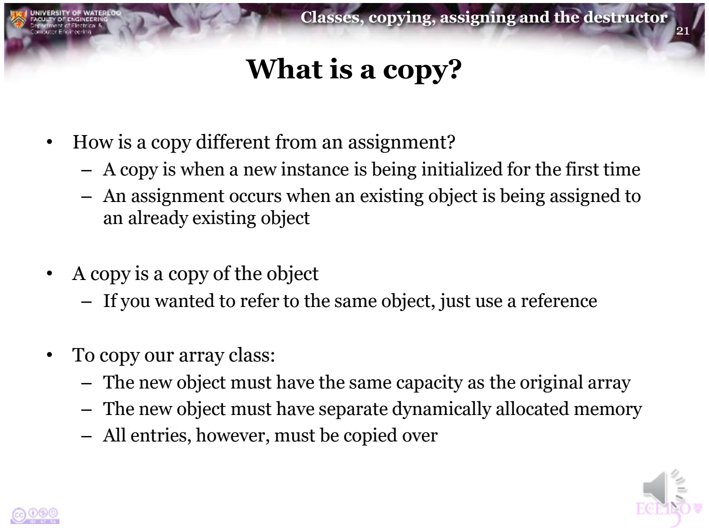


Copying and assigning

- In general, if your class dynamically allocates memory, start here:

```
class Class_name {
    public:
        Class_name(…);                // Always define a constructor
        // Do not copy or move...
        Class_name( Class_name const  &original ) = delete;
        Class_name( Class_name       &&original ) = delete;
       ~Classname();
        // Do not assign or move...
        Class_name &operator=( Class_name const  &rhs ) = delete;
        Class_name &operator=( Class_name       &&rhs ) = delete;
        // Other public member functions
    private:
        // Private member variables and
        // private helper member functions
};
```

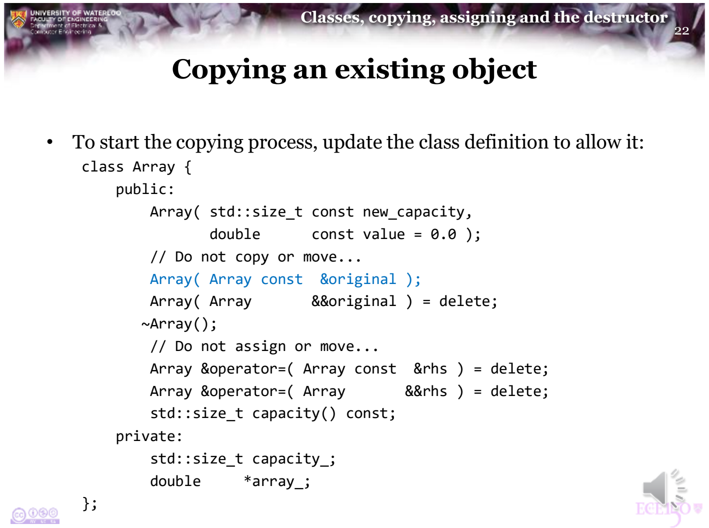
20



What is a copy?

- How is a copy different from an assignment?
  - A copy is when a new instance is being initialized for the first time
  - An assignment occurs when an existing object is being assigned to an already existing object
- A copy is a copy of the object
  - If you wanted to refer to the same object, just use a reference
- To copy our array class:
  - The new object must have the same capacity as the original array
  - The new object must have separate dynamically allocated memory
  - All entries, however, must be copied over

21

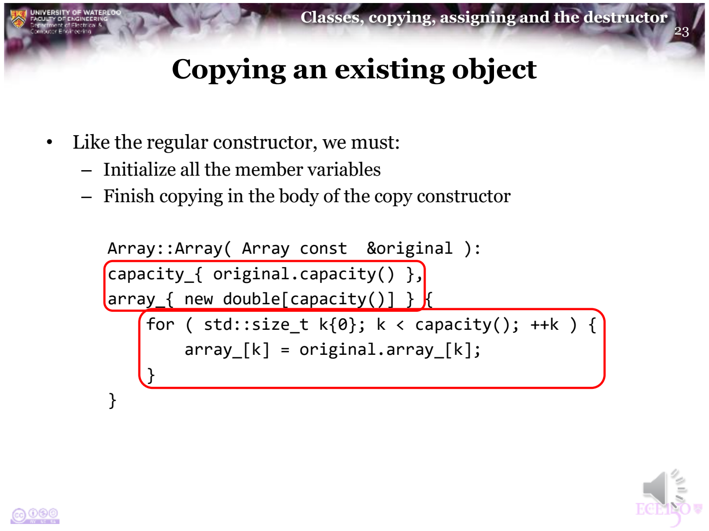


Copying an existing object

- To start the copying process, update the class definition to allow it:

```
class Array {
    public:
        Array( std::size_t const new_capacity,
               double      const value = 0.0 );
        // Do not copy or move...
        Array( Array const  &original );
        Array( Array       &&original ) = delete;
       ~Array();
        // Do not assign or move...
        Array &operator=( Array const  &rhs ) = delete;
        Array &operator=( Array       &&rhs ) = delete;
        std::size_t capacity() const;
    private:
        std::size_t capacity_;
        double     *array_;
};
```

22

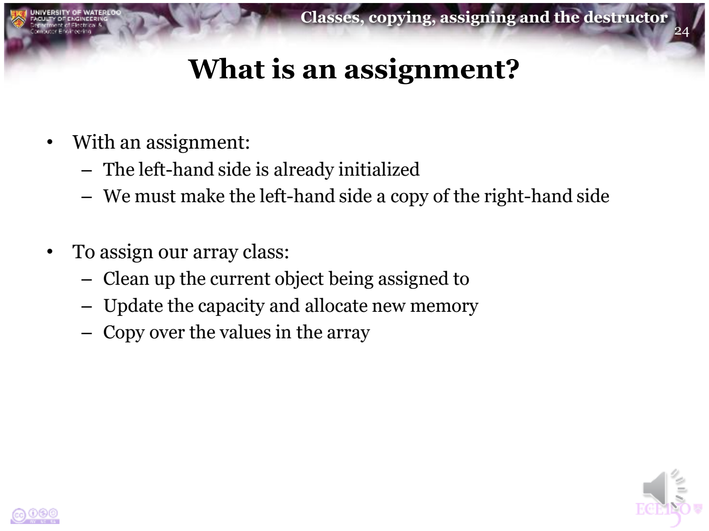


Copying an existing object

- Like the regular constructor, we must:
  - Initialize all the member variables
  - Finish copying in the body of the copy constructor

```
Array::Array( Array const  &original ):
capacity_{ original.capacity() },
array_{ new double[capacity()] } {
    for ( std::size_t k{0}; k < capacity(); ++k ) {
        array_[k] = original.array_[k];
    }
}
```

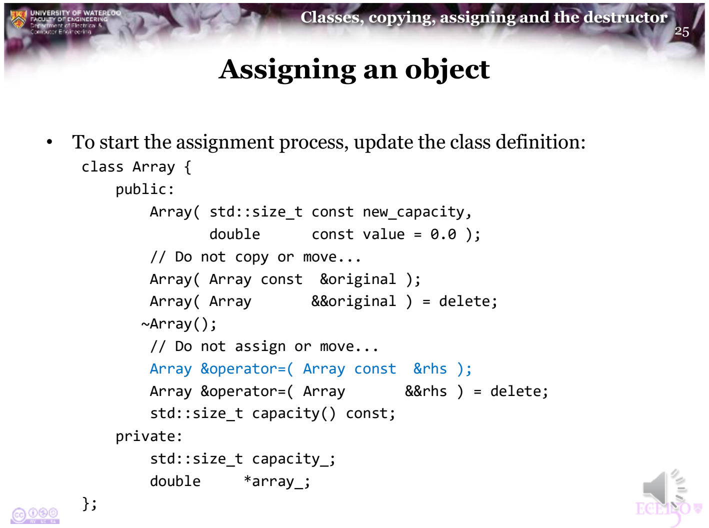
23



What is an assignment?

- With an assignment:
  - The left-hand side is already initialized
  - We must make the left-hand side a copy of the right-hand side
- To assign our array class:
  - Clean up the current object being assigned to
  - Update the capacity and allocate new memory
  - Copy over the values in the array

24

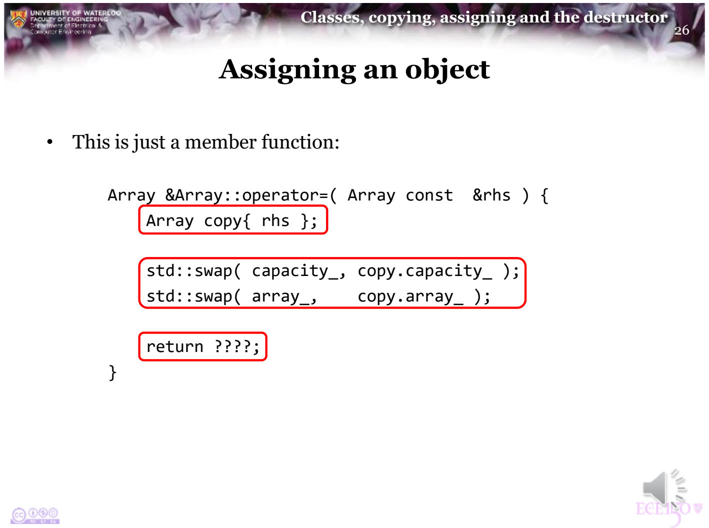


Assigning an object

- To start the assignment process, update the class definition:

```
class Array {
    public:
        Array( std::size_t const new_capacity,
               double      const value = 0.0 );
        // Do not copy or move...
        Array( Array const  &original );
        Array( Array       &&original ) = delete;
       ~Array();
        // Do not assign or move...
        Array &operator=( Array const  &rhs );
        Array &operator=( Array       &&rhs ) = delete;
        std::size_t capacity() const;
    private:
        std::size_t capacity_;
        double     *array_;
};
```

25

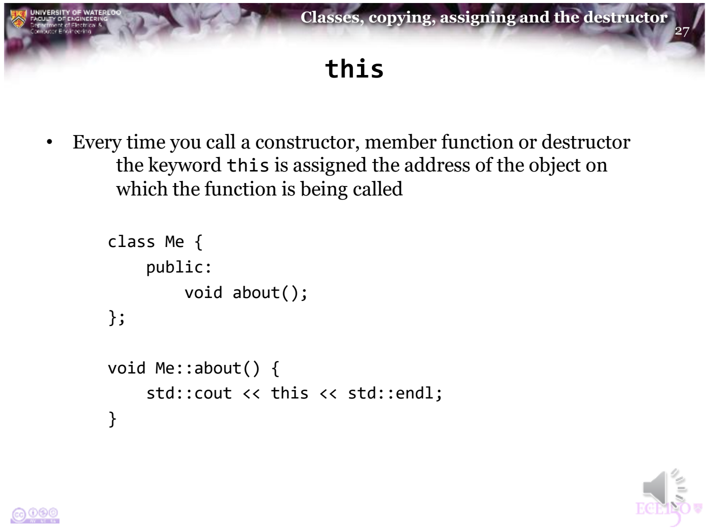


Assigning an object

- This is just a member function:

```
Array &Array::operator=( Array const  &rhs ) {
    Array copy{ rhs };

    std::swap( capacity_, copy.capacity_ );
    std::swap( array_,    copy.array_ );

    return ????;
}
```

26

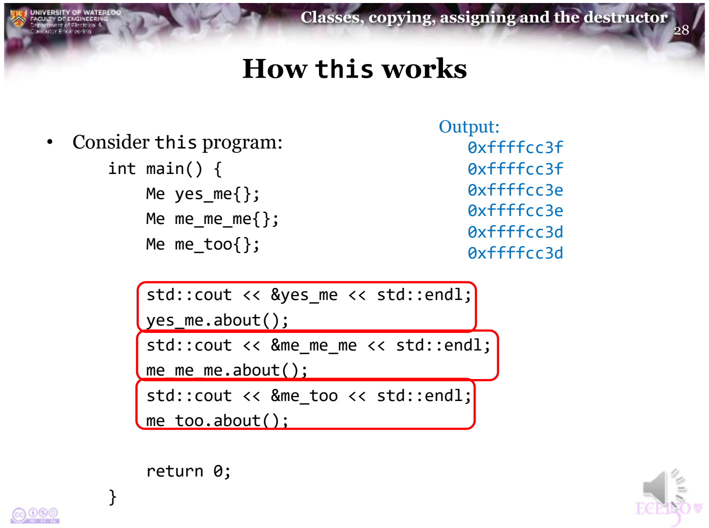


this

- Every time you call a constructor, member function or destructor the keyword this is assigned the address of the object on which the function is being called

```
class Me {
    public:
        void about();
};

void Me::about() {
    std::cout << this << std::endl;
}
```

27



How this works

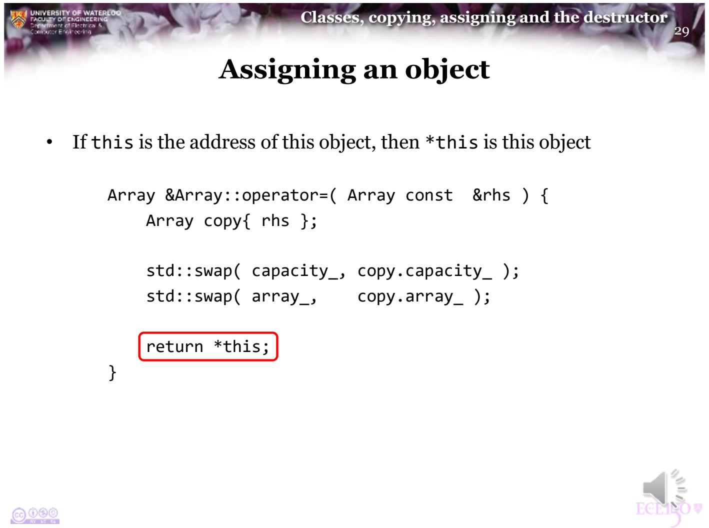
- Consider this program:

```
int main() {
    Me yes_me{};
    Me me_me_me{};
    Me me_too{};

    std::cout << &yes_me << std::endl;
    yes_me.about();
    std::cout << &me_me_me << std::endl;
    me_me_me.about();
    std::cout << &me_too << std::endl;
    me_too.about();

    return 0;
}
```

Output:

```
0xffffcc3f
0xffffcc3f
0xffffcc3e
0xffffcc3e
0xffffcc3d
0xffffcc3d
```

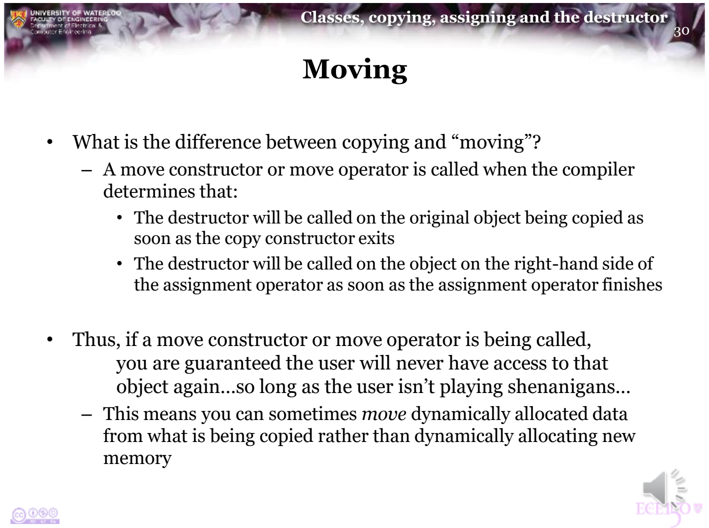
28



Assigning an object

- If `this` is the address of this object, then `*this` is this object

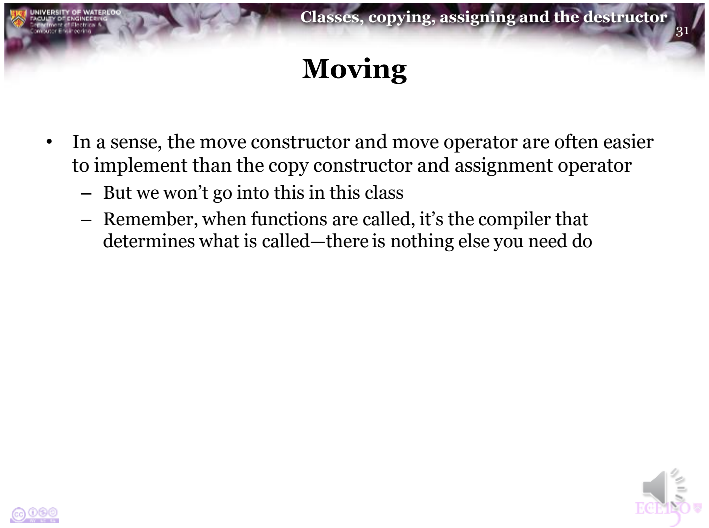
```
Array &Array::operator=( Array const  &rhs ) {
    Array copy{ rhs };

    std::swap( capacity_, copy.capacity_ );
    std::swap( array_,    copy.array_ );

    return *this;
}
```



Moving

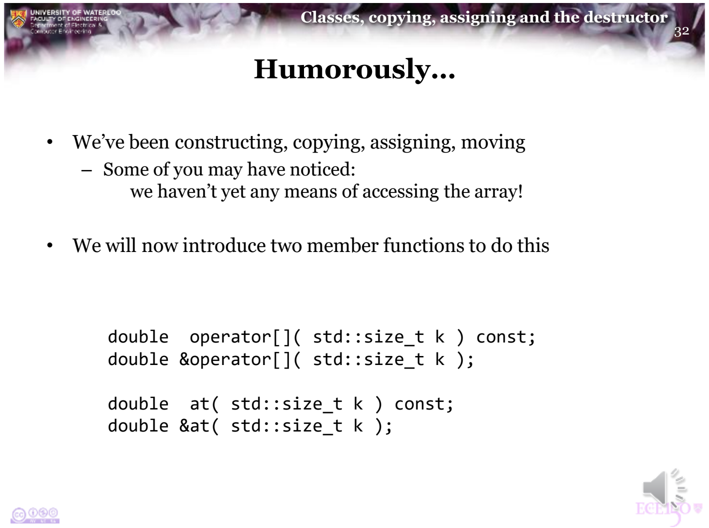
- What is the difference between copying and “moving”?
  - A move constructor or move operator is called when the compiler determines that:
    - The destructor will be called on the original object being copied as soon as the copy constructor exits
    - The destructor will be called on the object on the right-hand side of the assignment operator as soon as the assignment operator finishes

- Thus, if a move constructor or move operator is being called, you are guaranteed the user will never have access to that object again…so long as the user isn’t playing shenanigans…
  - This means you can sometimes *move* dynamically allocated data from what is being copied rather than dynamically allocating new memory



Moving

- In a sense, the move constructor and move operator are often easier to implement than the copy constructor and assignment operator
  - But we won’t go into this in this class
  - Remember, when functions are called, it’s the compiler that determines what is called—there is nothing else you need do

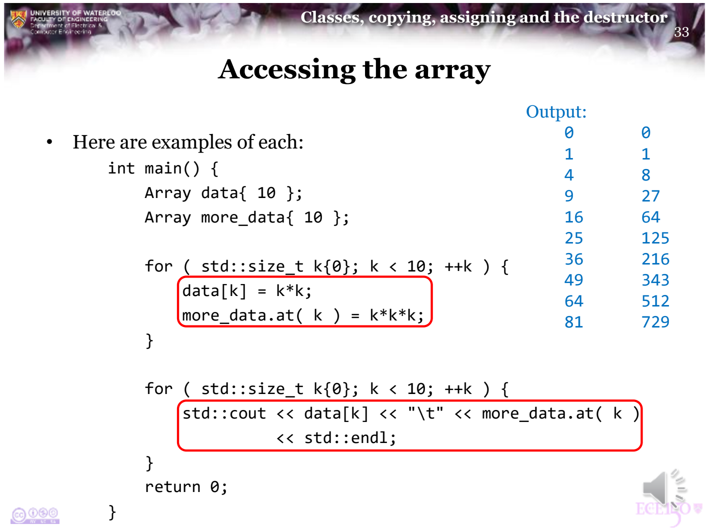


Humorously…

- We’ve been constructing, copying, assigning, moving
  - Some of you may have noticed:
    we haven’t yet any means of accessing the array!

- We will now introduce two member functions to do this

```
double  operator[]( std::size_t k ) const;
double &operator[]( std::size_t k );

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



Classes, copying, assigning and the destructor 33

Accessing the array

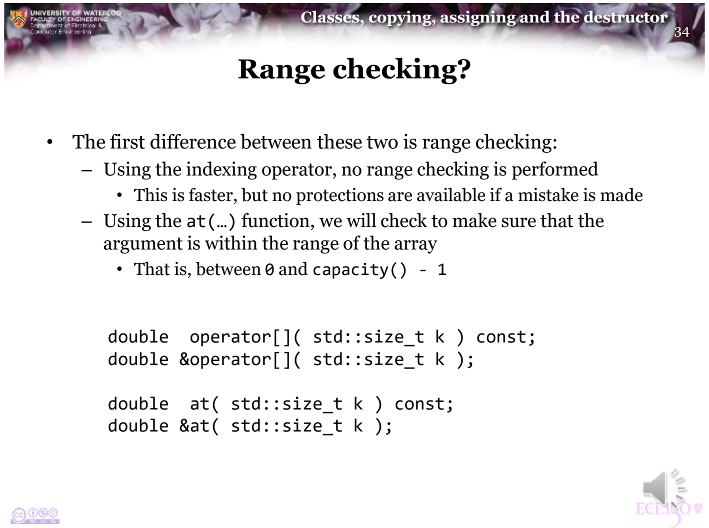
- Here are examples of each:

```
int main() {
    Array data{ 10 };
    Array more_data{ 10 };

    for ( std::size_t k{0}; k < 10; ++k ) {
        data[k] = k*k;
        more_data.at( k ) = k*k*k;
    }

    for ( std::size_t k{0}; k < 10; ++k ) {
        std::cout << data[k] << "\t" << more_data.at( k )
                  << std::endl;
    }
    return 0;
}
```

Output:

```
0     0
1     1
4     8
9     27
16    64
25    125
36    216
49    343
64    512
81    729
```

33



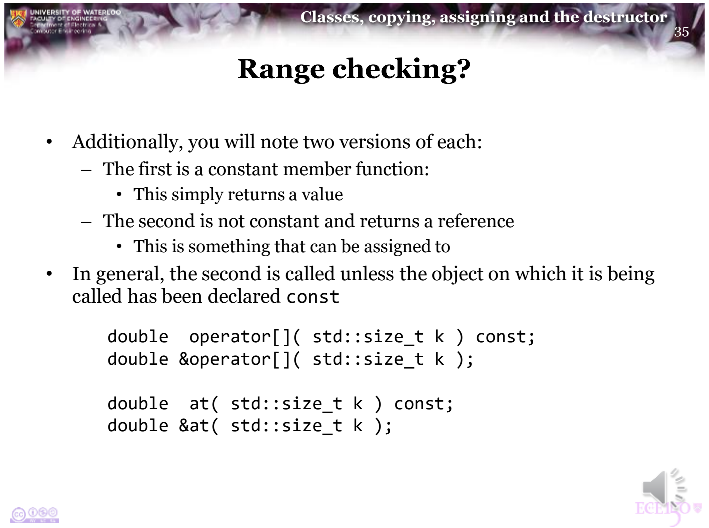
Classes, copying, assigning and the destructor 34

Range checking?

- The first difference between these two is range checking:
  - Using the indexing operator, no range checking is performed
    - This is faster, but no protections are available if a mistake is made
  - Using the `at(…)` function, we will check to make sure that the argument is within the range of the array
    - That is, between `0` and `capacity() - 1`

```
double  operator[]( std::size_t k ) const;
double &operator[]( std::size_t k );

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

34



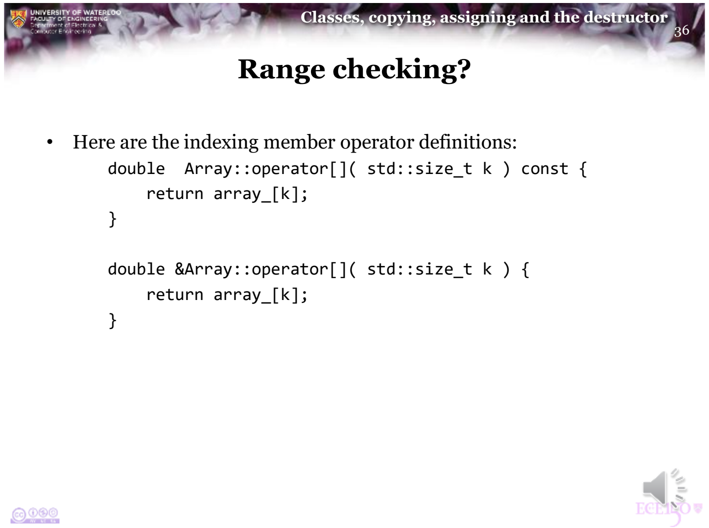
Classes, copying, assigning and the destructor 35

Range checking?

- Additionally, you will note two versions of each:
  - The first is a constant member function:
    - This simply returns a value
  - The second is not constant and returns a reference
    - This is something that can be assigned to
- In general, the second is called unless the object on which it is being called has been declared `const`

```
double  operator[]( std::size_t k ) const;
double &operator[]( std::size_t k );

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

35



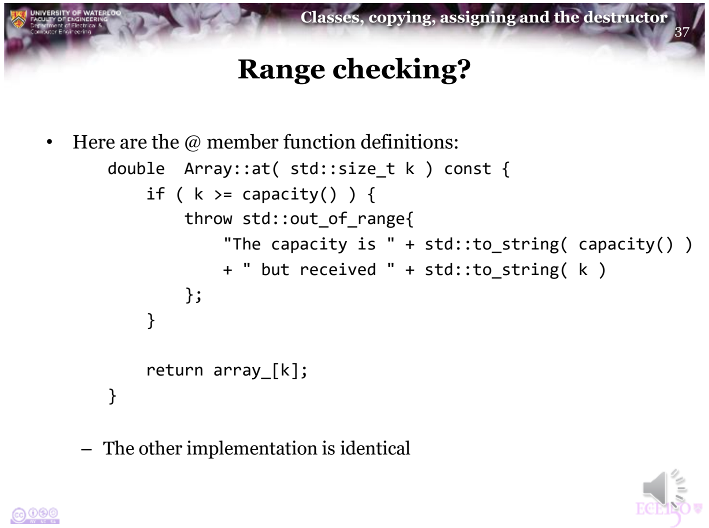
Classes, copying, assigning and the destructor 36

Range checking?

- Here are the indexing member operator definitions:

```
double  Array::operator[]( std::size_t k ) const {
    return array_[k];
}

double &Array::operator[]( std::size_t k ) {
    return array_[k];
}
```

36



Classes, copying, assigning and the destructor 37

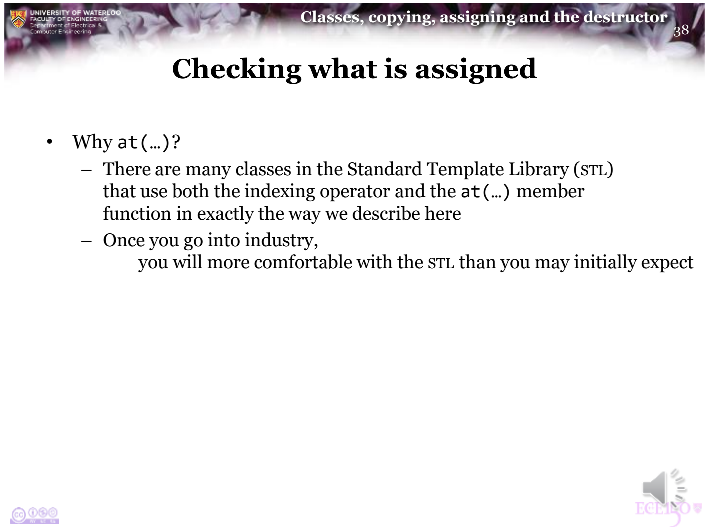
Range checking?

- Here are the @ member function definitions:

```
double  Array::at( std::size_t k ) const {
    if ( k >= capacity() ) {
        throw std::out_of_range{
            "The capacity is " + std::to_string( capacity() )
            + " but received " + std::to_string( k )
        };
    }

    return array_[k];
}
```

  - The other implementation is identical

37

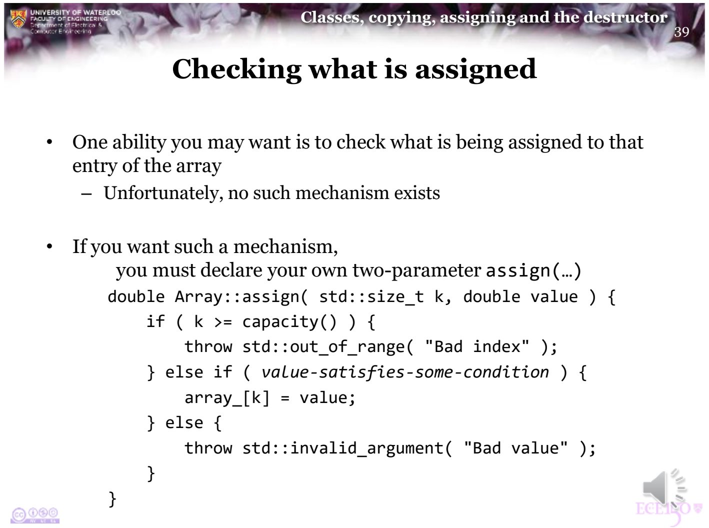


Classes, copying, assigning and the destructor 38

Checking what is assigned

- Why `at(…)`?
  - There are many classes in the Standard Template Library (STL) that use both the indexing operator and the `at(…)` member function in exactly the way we describe here
  - Once you go into industry,
    you will more comfortable with the STL than you may initially expect

38



Classes, copying, assigning and the destructor 39

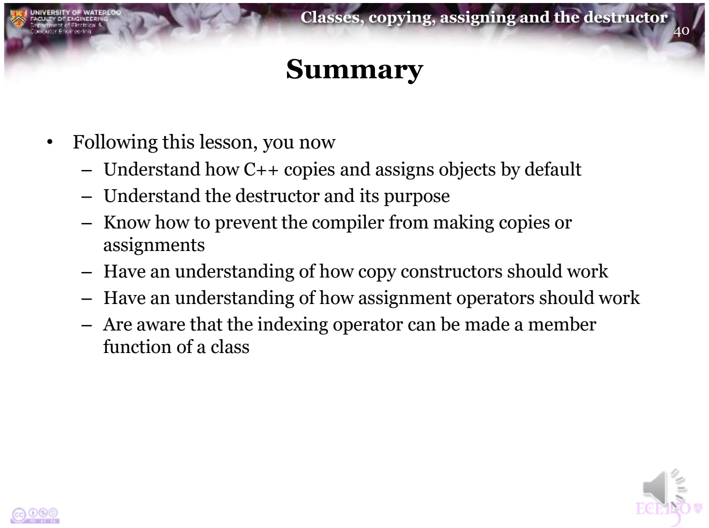
Checking what is assigned

- One ability you may want is to check what is being assigned to that entry of the array
  - Unfortunately, no such mechanism exists

- If you want such a mechanism,
  you must declare your own two-parameter `assign(…)`

```
double Array::assign( std::size_t k, double value ) {
    if ( k >= capacity() ) {
        throw std::out_of_range( "Bad index" );
    } else if ( value-satisfies-some-condition ) {
        array_[k] = value;
    } else {
        throw std::invalid_argument( "Bad value" );
    }
}
```

39

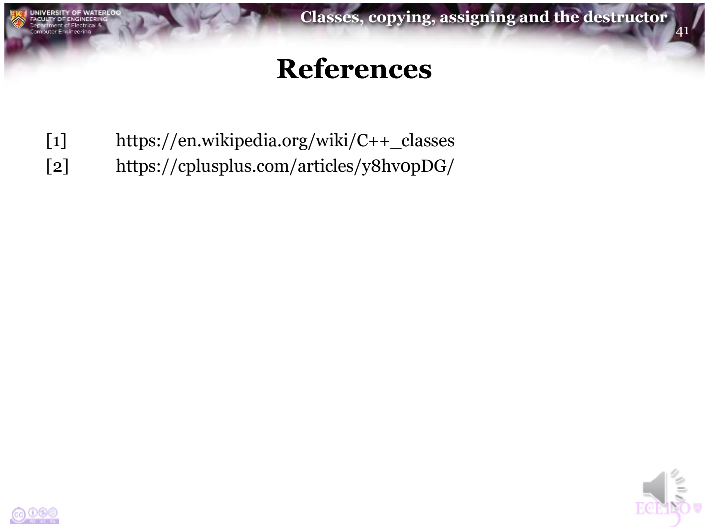


Classes, copying, assigning and the destructor 40

Summary

- Following this lesson, you now
  - Understand how C++ copies and assigns objects by default
  - Understand the destructor and its purpose
  - Know how to prevent the compiler from making copies or assignments
  - Have an understanding of how copy constructors should work
  - Have an understanding of how assignment operators should work
  - Are aware that the indexing operator can be made a member function of a class

40

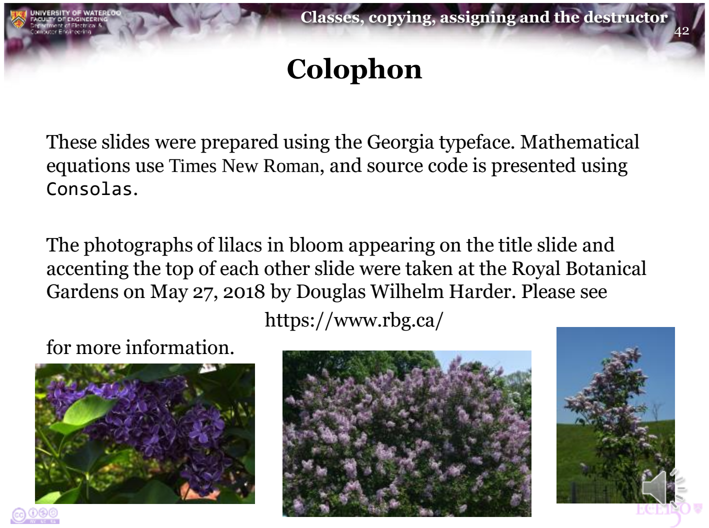


References

[1] https://en.wikipedia.org/wiki/C++_classes

[2] https://cplusplus.com/articles/y8hv0pDG/

41



Colophon

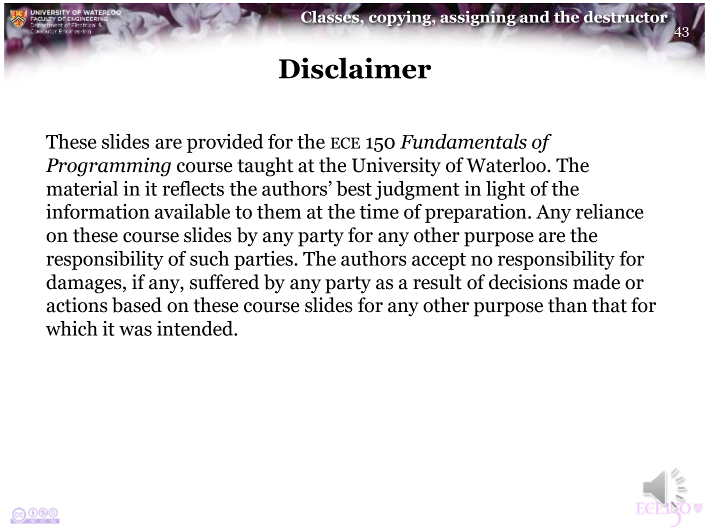
These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

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.

42



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.

43