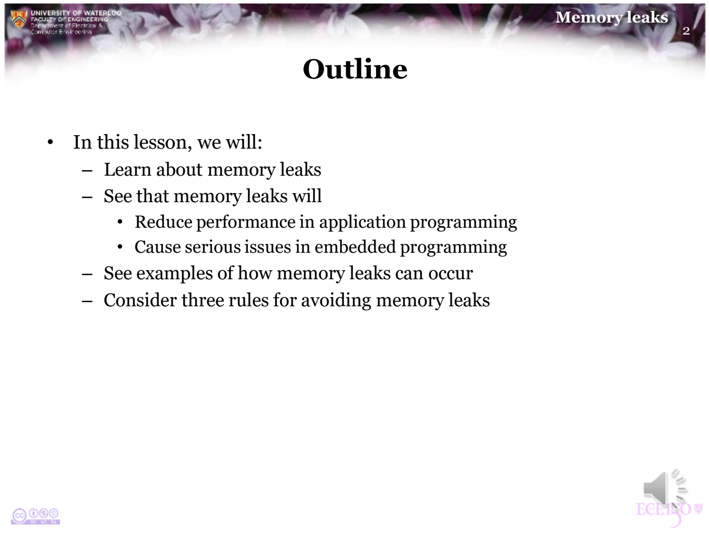
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

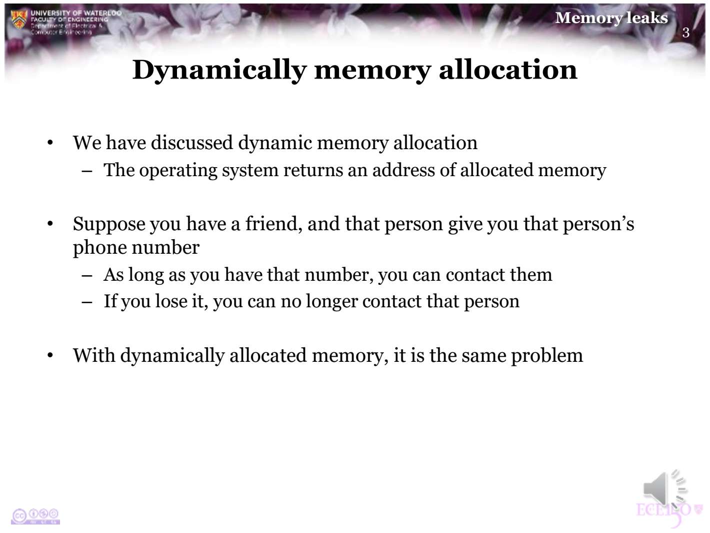
Memory leaks

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



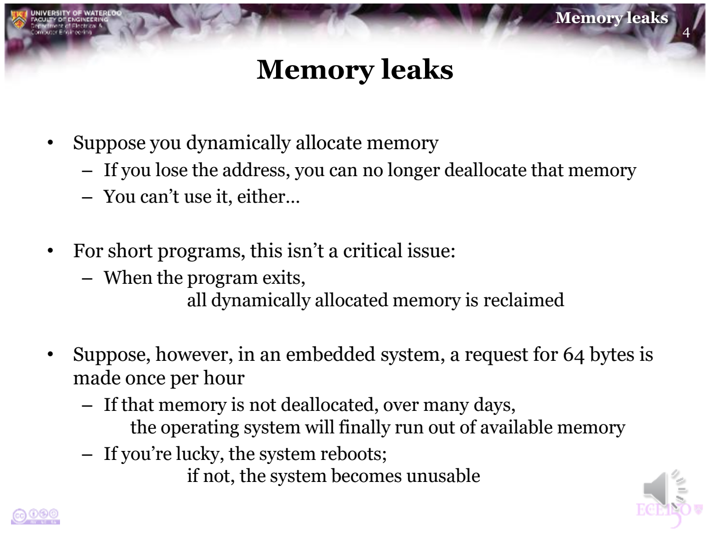
Outline

- In this lesson, we will:
  - Learn about memory leaks
  - See that memory leaks will
    - Reduce performance in application programming
    - Cause serious issues in embedded programming
  - See examples of how memory leaks can occur
  - Consider three rules for avoiding memory leaks



Dynamically memory allocation

- We have discussed dynamic memory allocation
  - The operating system returns an address of allocated memory
- Suppose you have a friend, and that person give you that person's phone number
  - As long as you have that number, you can contact them
  - If you lose it, you can no longer contact that person
- With dynamically allocated memory, it is the same problem



Memory leaks

- Suppose you dynamically allocate memory
  - If you lose the address, you can no longer deallocate that memory
  - You can't use it, either…
- For short programs, this isn't a critical issue:
  - When the program exits,
    all dynamically allocated memory is reclaimed
- Suppose, however, in an embedded system, a request for 64 bytes is made once per hour
  - If that memory is not deallocated, over many days,
    the operating system will finally run out of available memory
  - If you're lucky, the system reboots;
    if not, the system becomes unusable

Losing scope

- Memory leaks occur if all pointers go out of scope:

```
int f( parameters... ) {
    double *a_data{ new double[10]{} };
    // Do something with the allocated memory

    return return-value;
}
```

If the memory is only required in this function,
call `delete[]` on it

If the memory is required elsewhere,
return the address or store that address elsewhere

Losing scope

- Sometimes it may be more subtle:

```
int f( parameters... ) {
    double *a_data{new double[10]{}};
    while ( condition ) {
        // Do something with the allocated memory

        if ( another-condition ) {
            return return-value;  // memory not deallocated
        }
    }

    delete[] a_data;
    return return-value;
}
```

Losing scope

- Some programming guidelines require only one return per function specifically to avoid this—cleanup happens once ans is guaranteed

```
int f( parameters... ) {
    bool finished{ false };
    double *a_data{new double[10]{}};
    while ( condition && !finished ) {
        // Do something with the allocated memory

        if ( another-condition ) {
            finished = true;
        }
    }

    delete[] a_data;
    return return-value;
}
```

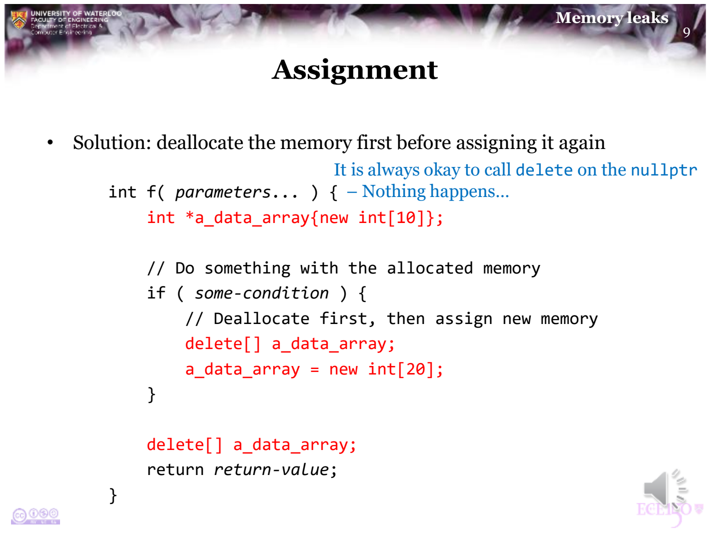
Assignment

- Memory leaks occur if pointers are overwritten without the memory first being deallocated

```
int f( parameters... ) {
    int *a_data{new int[10]};

    // Do something with the allocated memory
    if ( some-condition ) {
        // We need a larger array
        a_data = new int[20];
    }

    delete[] a_data;
    return return-value;
}
```



Assignment

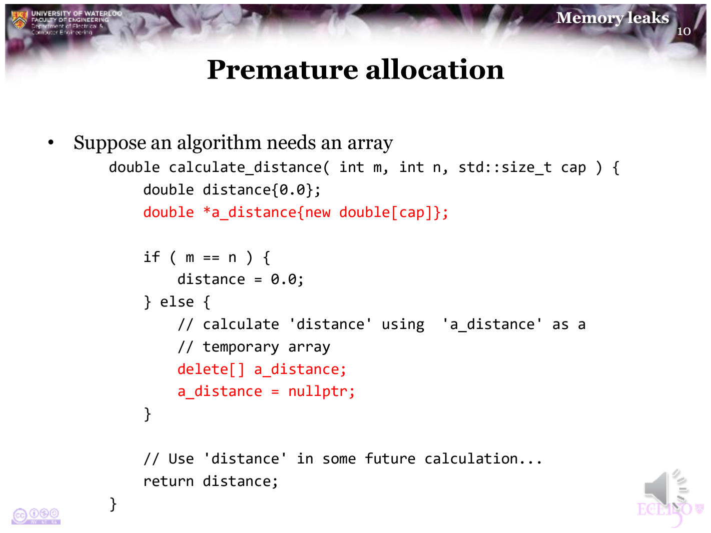
- Solution: deallocate the memory first before assigning it again

It is always okay to call `delete` on the `nullptr` – Nothing happens...

```
int f( parameters... ) {
    int *a_data_array{new int[10]};

    // Do something with the allocated memory
    if ( some-condition ) {
        // Deallocate first, then assign new memory
        delete[] a_data_array;
        a_data_array = new int[20];
    }

    delete[] a_data_array;
    return return-value;
}
```



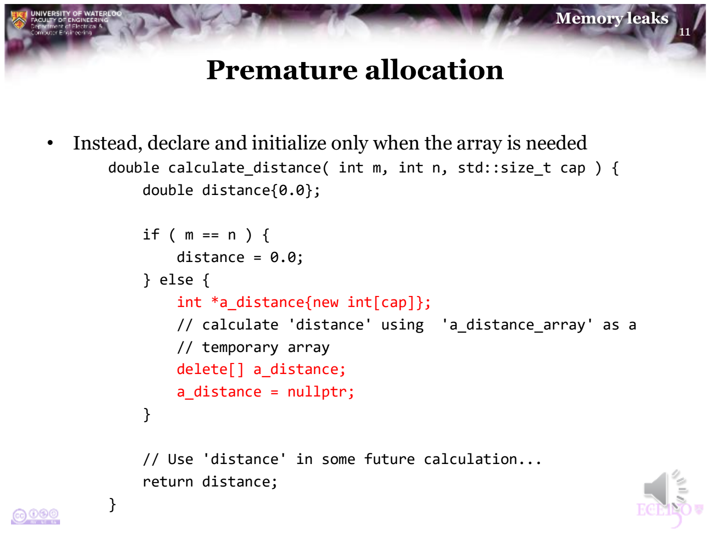
Premature allocation

- Suppose an algorithm needs an array

```
double calculate_distance( int m, int n, std::size_t cap ) {
    double distance{0.0};
    double *a_distance{new double[cap]};

    if ( m == n ) {
        distance = 0.0;
    } else {
        // calculate 'distance' using  'a_distance' as a
        // temporary array
        delete[] a_distance;
        a_distance = nullptr;
    }

    // Use 'distance' in some future calculation...
    return distance;
}
```



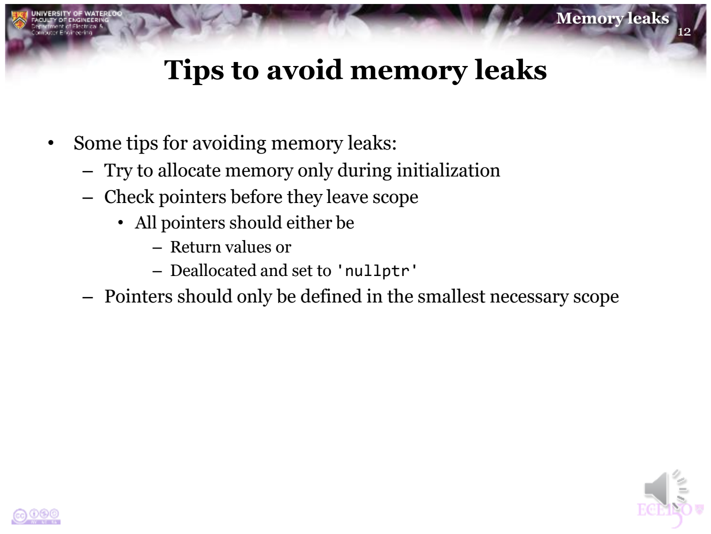
Premature allocation

- Instead, declare and initialize only when the array is needed

```
double calculate_distance( int m, int n, std::size_t cap ) {
    double distance{0.0};

    if ( m == n ) {
        distance = 0.0;
    } else {
        int *a_distance{new int[cap]};
        // calculate 'distance' using  'a_distance_array' as a
        // temporary array
        delete[] a_distance;
        a_distance = nullptr;
    }

    // Use 'distance' in some future calculation...
    return distance;
}
```



Tips to avoid memory leaks

- Some tips for avoiding memory leaks:
  - Try to allocate memory only during initialization
  - Check pointers before they leave scope
    - All pointers should either be
      - Return values or
      - Deallocated and set to `'nullptr'`
  - Pointers should only be defined in the smallest necessary scope

Restrict allocation to initialization

- Try to restrict allocation to initialization
  - If possibly, only allocate when a variable is being initialized

```
double *function_name( parameters… ) {
    // do not declare a_int here

    while ( condition ) {
        int *p_int{new int{some_value}};
        // Do something with 'p_int'
        delete p_int;
        p_int = nullptr;

        // Do something else...
        assert( p_int == nullptr );
    }
}
```

Check pointers before they leave scope

- Avoiding memory leaks requires rigorous control on the assignment:
  - After deleting a pointer, set the value to `nullptr`
  - At the end of a function,
    either all pointers should be `nullptr` or returned

```
double *function_name( parameters… ) {
    int *p_int1{new int{42}};
    int *p_int2{ p_int };
    int *a_return_value{new double[10]};

     // Do something...
    //  - at some point you must delete 'p_int1' and
    //    assign it the value 'nullptr'

    assert( p_int1 == nullptr );
    assert( p_int2 == nullptr );
    return a_return_value;
}
```

Use the smallest possible scope

- Use the smallest scope necessary
  - If you only need a pointer in a block,
    don't declare it outside that block

```
double *function_name( parameters… ) {
    // Do not declare 'a_vec' here...

    while ( condition ) {
        // Do something...

        if ( special-condition ) {
            double *a_vec{new double[3]};

            // Use 'a_vec' and at some point, delete[] it
            // and set 'a_vec' to nullptr...

            assert( a_vec == nullptr );
        }
    }
}
```

Summary

- Following this lesson, you now
  - Understand dynamic memory allocation can result in leaks
  - Know that leaks are critically dangerous in embedded systems
  - Know you must ensure you track the address and deallocate the memory when it is finished
  - Understand that you should
    - Restrict allocation to initialization
    - Check pointers before they go out of scope
    - Restrict pointers to the smallest possible scope

References

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

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