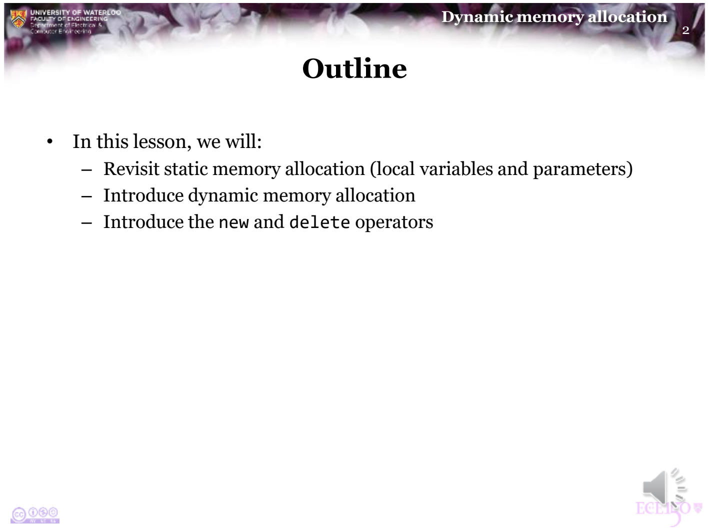
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

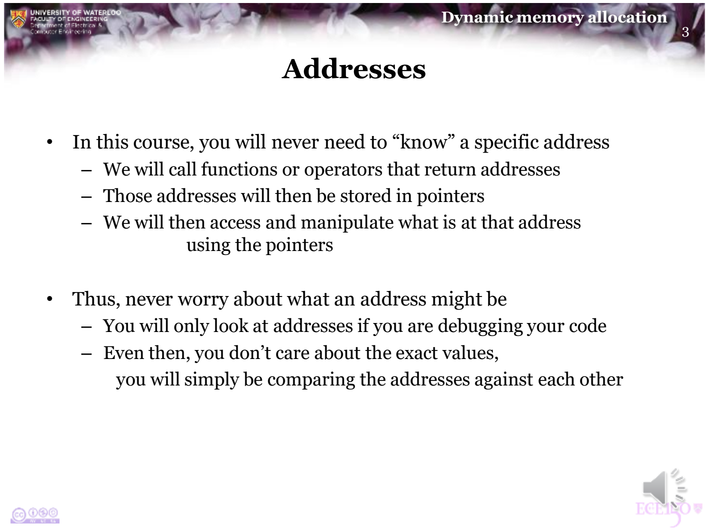
Dynamic memory allocation

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



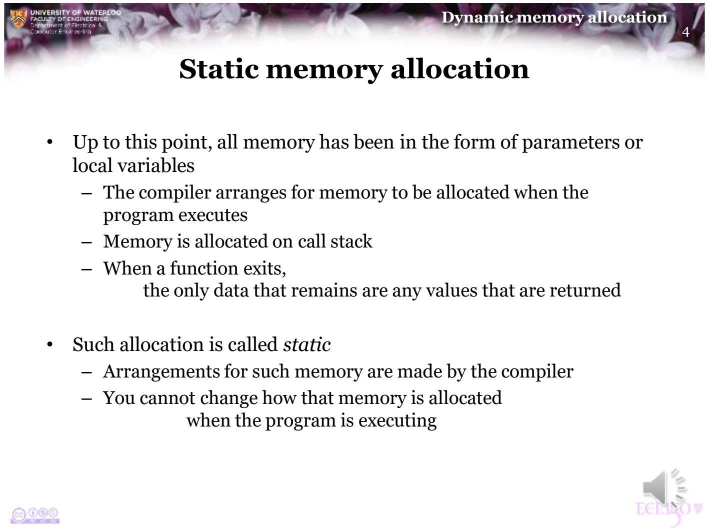
Outline

- In this lesson, we will:
  - Revisit static memory allocation (local variables and parameters)
  - Introduce dynamic memory allocation
  - Introduce the `new` and `delete` operators



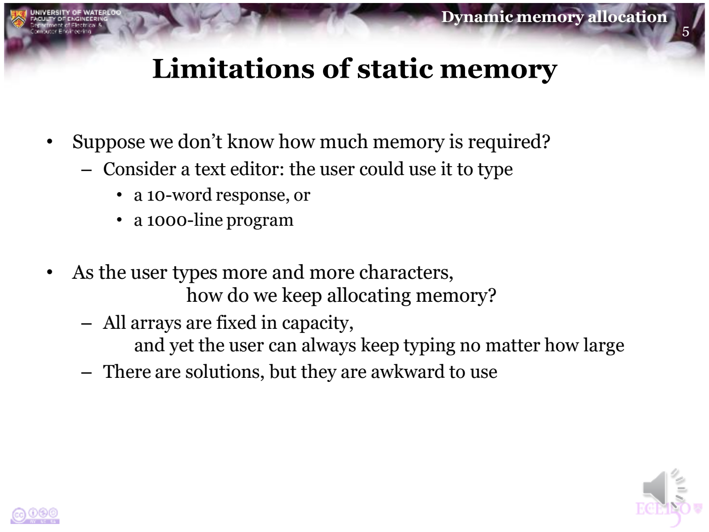
Addresses

- In this course, you will never need to “know” a specific address
  - We will call functions or operators that return addresses
  - Those addresses will then be stored in pointers
  - We will then access and manipulate what is at that address using the pointers
- Thus, never worry about what an address might be
  - You will only look at addresses if you are debugging your code
  - Even then, you don’t care about the exact values, you will simply be comparing the addresses against each other



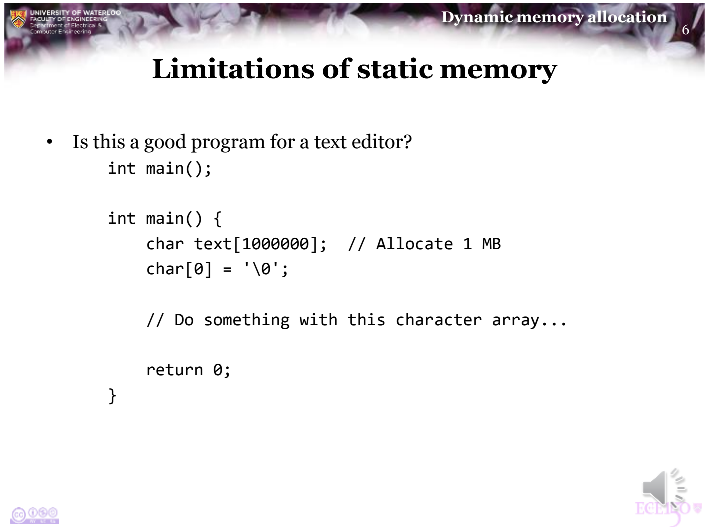
Static memory allocation

- Up to this point, all memory has been in the form of parameters or local variables
  - The compiler arranges for memory to be allocated when the program executes
  - Memory is allocated on call stack
  - When a function exits, the only data that remains are any values that are returned
- Such allocation is called *static*
  - Arrangements for such memory are made by the compiler
  - You cannot change how that memory is allocated when the program is executing



Limitations of static memory

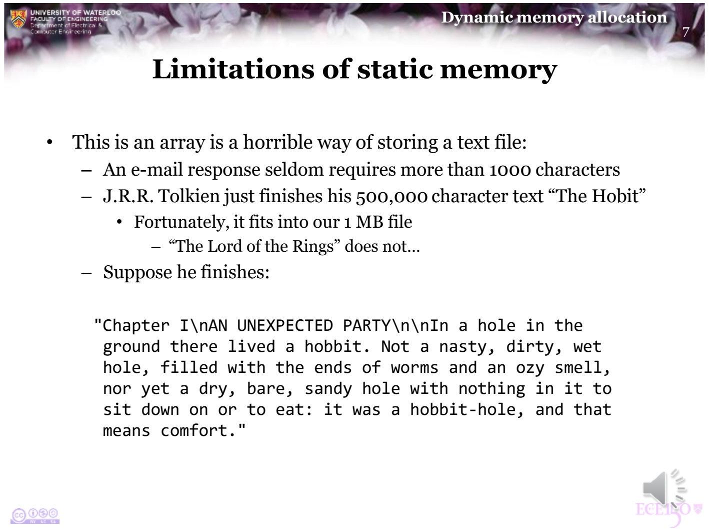
- Suppose we don't know how much memory is required?
  - Consider a text editor: the user could use it to type
    - a 10-word response, or
    - a 1000-line program
- As the user types more and more characters,
  how do we keep allocating memory?
  - All arrays are fixed in capacity,
    and yet the user can always keep typing no matter how large
  - There are solutions, but they are awkward to use



Limitations of static memory

- Is this a good program for a text editor?

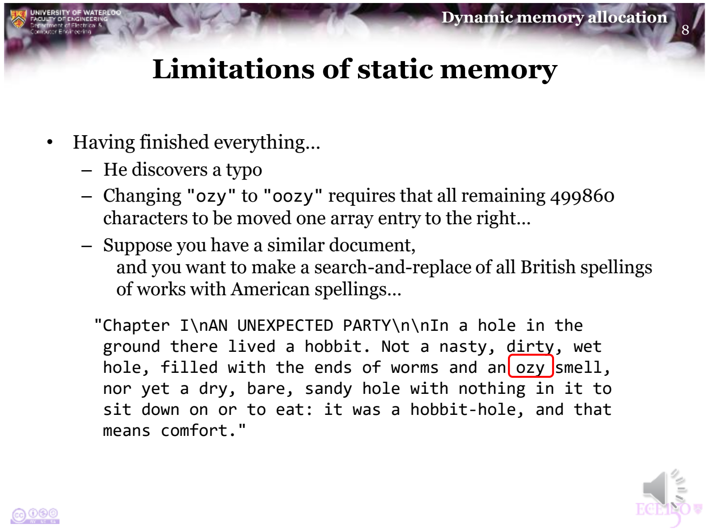
```
int main();

int main() {
    char text[1000000];  // Allocate 1 MB
    char[0] = '\0';

    // Do something with this character array...

    return 0;
}
```



Limitations of static memory

- This is an array is a horrible way of storing a text file:
  - An e-mail response seldom requires more than 1000 characters
  - J.R.R. Tolkien just finishes his 500,000 character text "The Hobit"
    - Fortunately, it fits into our 1 MB file
      - "The Lord of the Rings" does not...
  - Suppose he finishes:

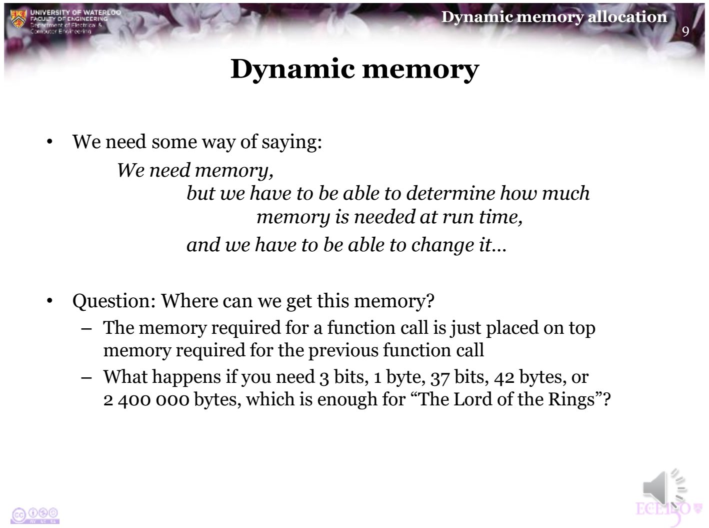
```
"Chapter I\nAN UNEXPECTED PARTY\n\nIn a hole in the
 ground there lived a hobbit. Not a nasty, dirty, wet
 hole, filled with the ends of worms and an ozy smell,
 nor yet a dry, bare, sandy hole with nothing in it to
 sit down on or to eat: it was a hobbit-hole, and that
 means comfort."
```



Limitations of static memory

- Having finished everything...
  - He discovers a typo
  - Changing "ozy" to "oozy" requires that all remaining 499860 characters to be moved one array entry to the right...
  - Suppose you have a similar document,
    and you want to make a search-and-replace of all British spellings of works with American spellings...

```
"Chapter I\nAN UNEXPECTED PARTY\n\nIn a hole in the
 ground there lived a hobbit. Not a nasty, dirty, wet
 hole, filled with the ends of worms and an ozy smell,
 nor yet a dry, bare, sandy hole with nothing in it to
 sit down on or to eat: it was a hobbit-hole, and that
 means comfort."
```

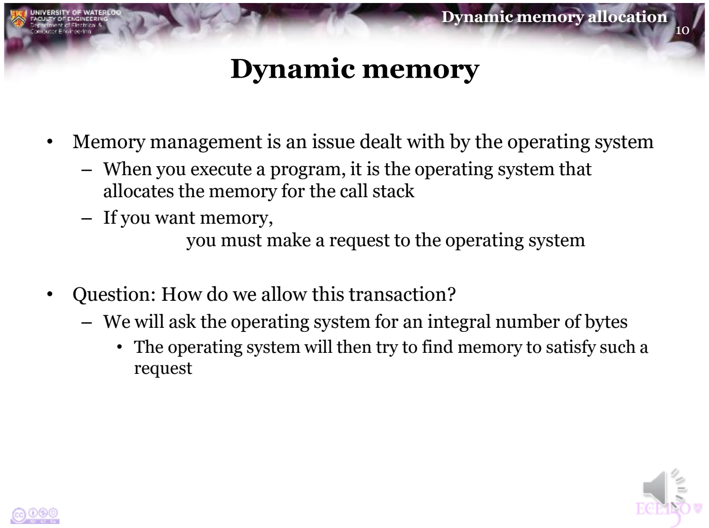


Dynamic memory

- We need some way of saying:

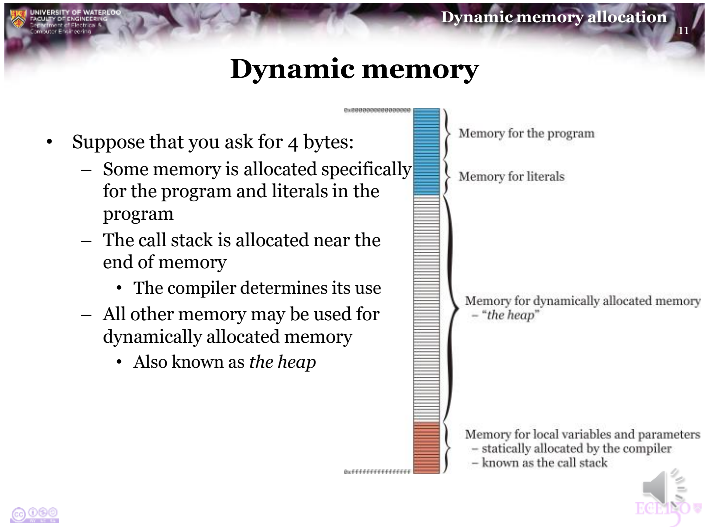
  We need memory,
  but we have to be able to determine how much memory is needed at run time,
  and we have to be able to change it…

- Question: Where can we get this memory?
  - The memory required for a function call is just placed on top memory required for the previous function call
  - What happens if you need 3 bits, 1 byte, 37 bits, 42 bytes, or 2 400 000 bytes, which is enough for "The Lord of the Rings"?



Dynamic memory

- Memory management is an issue dealt with by the operating system
  - When you execute a program, it is the operating system that allocates the memory for the call stack
  - If you want memory,
    you must make a request to the operating system
- Question: How do we allow this transaction?
  - We will ask the operating system for an integral number of bytes
    - The operating system will then try to find memory to satisfy such a request



Dynamic memory

- Suppose that you ask for 4 bytes:
  - Some memory is allocated specifically for the program and literals in the program
  - The call stack is allocated near the end of memory
    - The compiler determines its use
  - All other memory may be used for dynamically allocated memory
    - Also known as *the heap*

0x0000000000000000

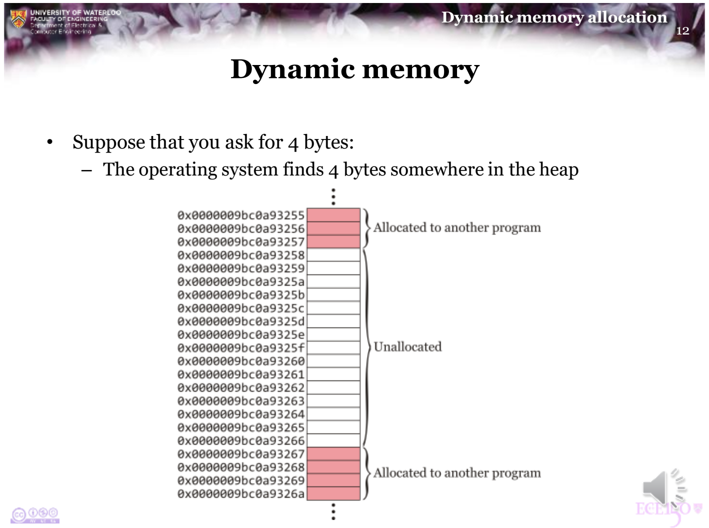
Memory for the program

Memory for literals

Memory for dynamically allocated memory
– *"the heap"*

Memory for local variables and parameters
– statically allocated by the compiler
– known as the call stack

0xffffffffffffffff



Dynamic memory

- Suppose that you ask for 4 bytes:
  - The operating system finds 4 bytes somewhere in the heap

```
0x0000009bc0a93255
0x0000009bc0a93256
0x0000009bc0a93257
0x0000009bc0a93258
0x0000009bc0a93259
0x0000009bc0a9325a
0x0000009bc0a9325b
0x0000009bc0a9325c
0x0000009bc0a9325d
0x0000009bc0a9325e
0x0000009bc0a9325f
0x0000009bc0a93260
0x0000009bc0a93261
0x0000009bc0a93262
0x0000009bc0a93263
0x0000009bc0a93264
0x0000009bc0a93265
0x0000009bc0a93266
0x0000009bc0a93267
0x0000009bc0a93268
0x0000009bc0a93269
0x0000009bc0a9326a
```

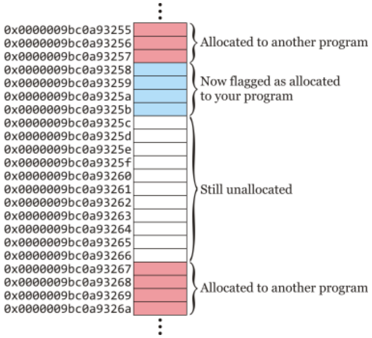
Allocated to another program

Unallocated

Allocated to another program

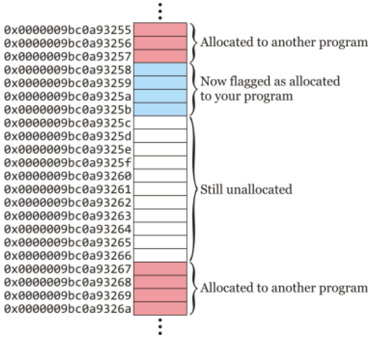
Dynamic memory

- The operating system flags these as belonging to your program
  - There are still 11 bytes left over, perhaps for some other request



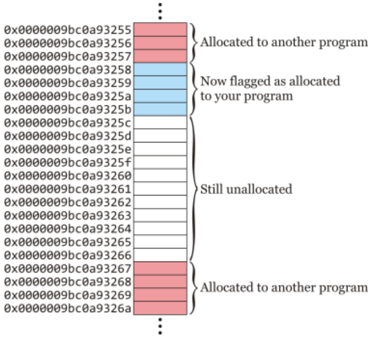
Dynamic memory

- How can you access this memory?
  - How does the operating system tell you that these 4 bytes are yours to use?



Dynamic memory

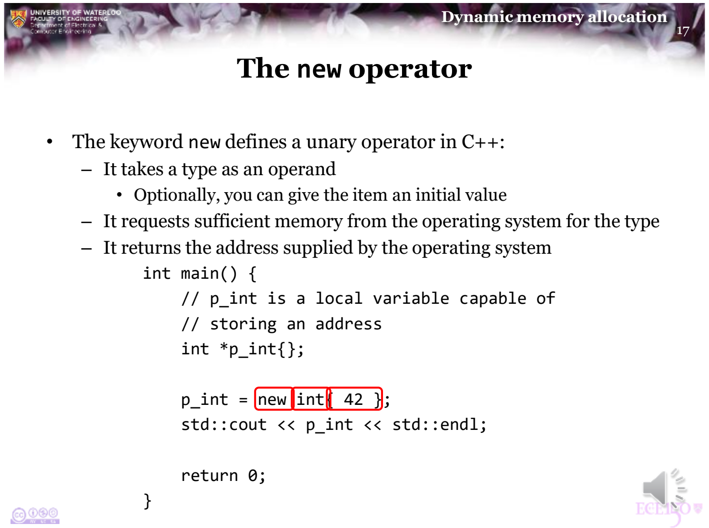
- A common solution is to return the address:
  - "Your 4 bytes are at memory location `0x000009bc0a93258`"



Dynamic memory

- The operating system could return this address, and we can assign this address to a pointer

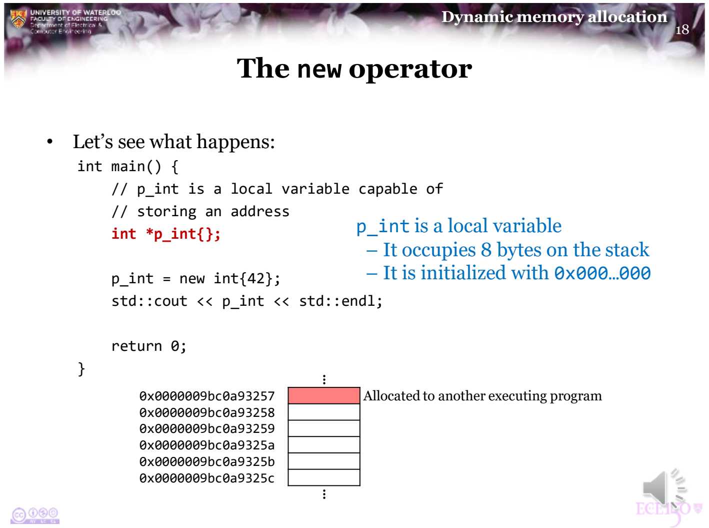
- Question: how many bytes do you need?
  - You could calculate it, but…C++ makes it easier
  - The compiler does the work



The new operator

- The keyword new defines a unary operator in C++:
  - It takes a type as an operand
    - Optionally, you can give the item an initial value
  - It requests sufficient memory from the operating system for the type
  - It returns the address supplied by the operating system

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{ 42 };
    std::cout << p_int << std::endl;

    return 0;
}
```



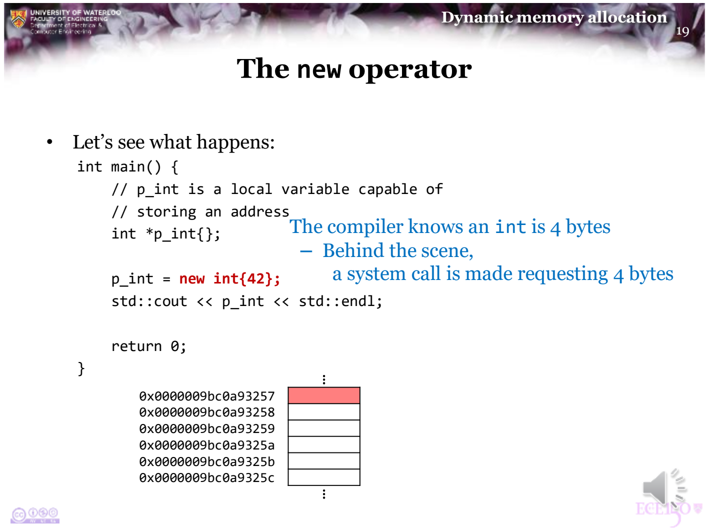
The new operator

- Let's see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

p_int is a local variable
- It occupies 8 bytes on the stack
- It is initialized with 0x000…000

```
                    ⋮
0x0000009bc0a93257 🟥  Allocated to another executing program
0x0000009bc0a93258 □
0x0000009bc0a93259 □
0x0000009bc0a9325a □
0x0000009bc0a9325b □
0x0000009bc0a9325c □
                    ⋮
```



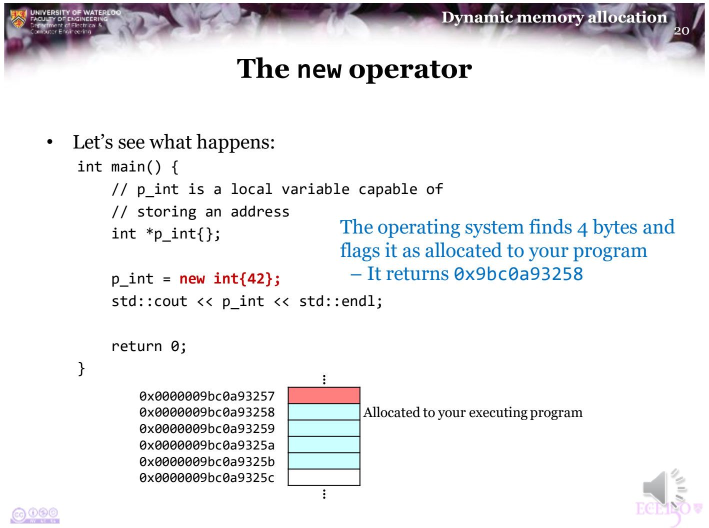
The new operator

- Let's see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

The compiler knows an int is 4 bytes
- Behind the scene,
  a system call is made requesting 4 bytes

```
                    ⋮
0x0000009bc0a93257 🟥
0x0000009bc0a93258 □
0x0000009bc0a93259 □
0x0000009bc0a9325a □
0x0000009bc0a9325b □
0x0000009bc0a9325c □
                    ⋮
```



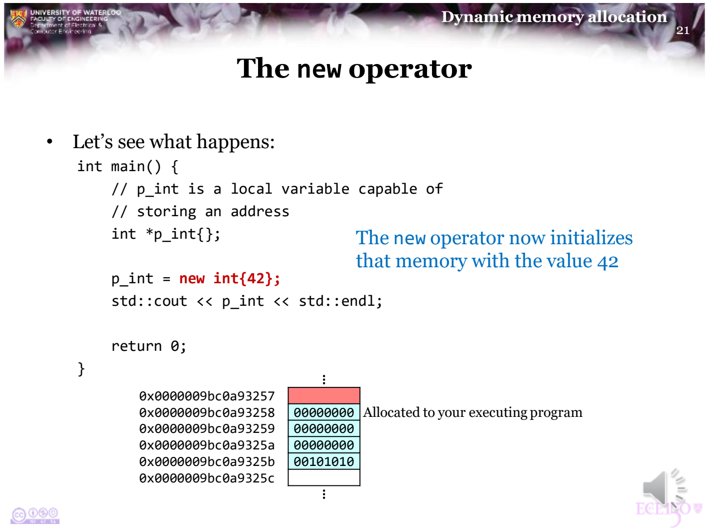
The new operator

- Let's see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

The operating system finds 4 bytes and flags it as allocated to your program
- It returns 0x9bc0a93258

```
                    ⋮
0x0000009bc0a93257 🟥
0x0000009bc0a93258 🟦  Allocated to your executing program
0x0000009bc0a93259 🟦
0x0000009bc0a9325a 🟦
0x0000009bc0a9325b 🟦
0x0000009bc0a9325c □
                    ⋮
```



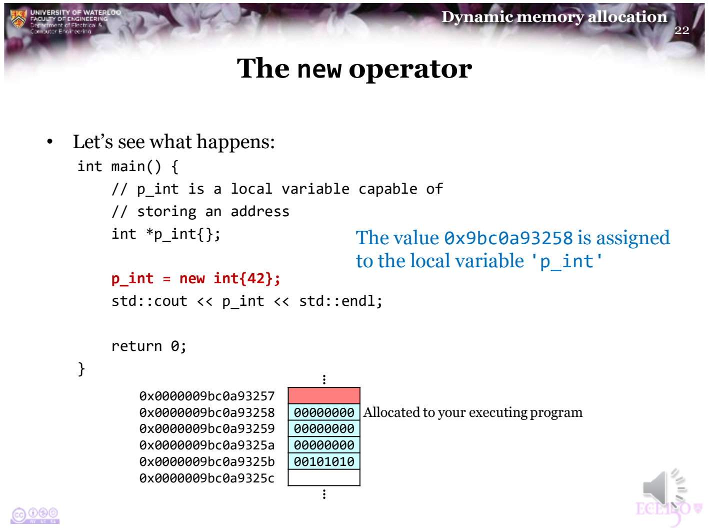
The new operator

- Let's see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

The new operator now initializes that memory with the value 42

```
                          ⋮
0x0000009bc0a93257
0x0000009bc0a93258   00000000  Allocated to your executing program
0x0000009bc0a93259   00000000
0x0000009bc0a9325a   00000000
0x0000009bc0a9325b   00101010
0x0000009bc0a9325c
                          ⋮
```



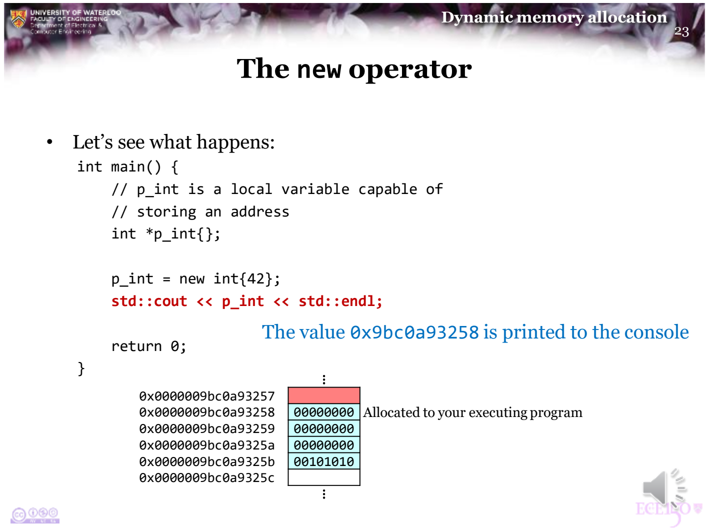
The new operator

- Let's see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

The value `0x9bc0a93258` is assigned to the local variable `'p_int'`

```
                          ⋮
0x0000009bc0a93257
0x0000009bc0a93258   00000000  Allocated to your executing program
0x0000009bc0a93259   00000000
0x0000009bc0a9325a   00000000
0x0000009bc0a9325b   00101010
0x0000009bc0a9325c
                          ⋮
```



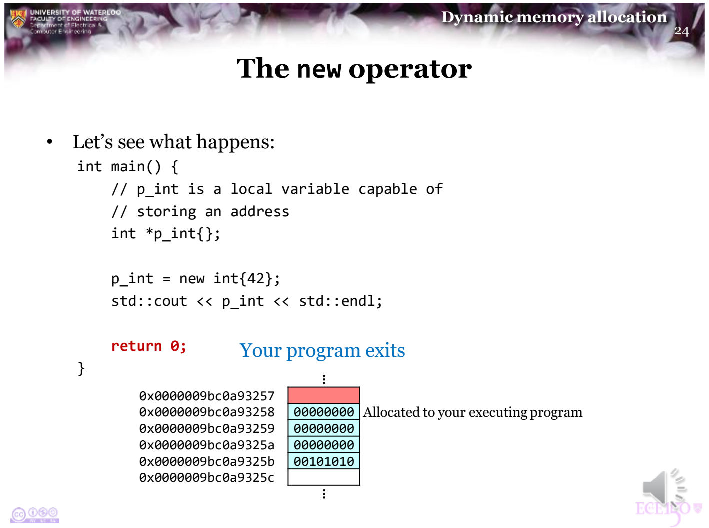
The new operator

- Let's see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

The value `0x9bc0a93258` is printed to the console

```
                          ⋮
0x0000009bc0a93257
0x0000009bc0a93258   00000000  Allocated to your executing program
0x0000009bc0a93259   00000000
0x0000009bc0a9325a   00000000
0x0000009bc0a9325b   00101010
0x0000009bc0a9325c
                          ⋮
```



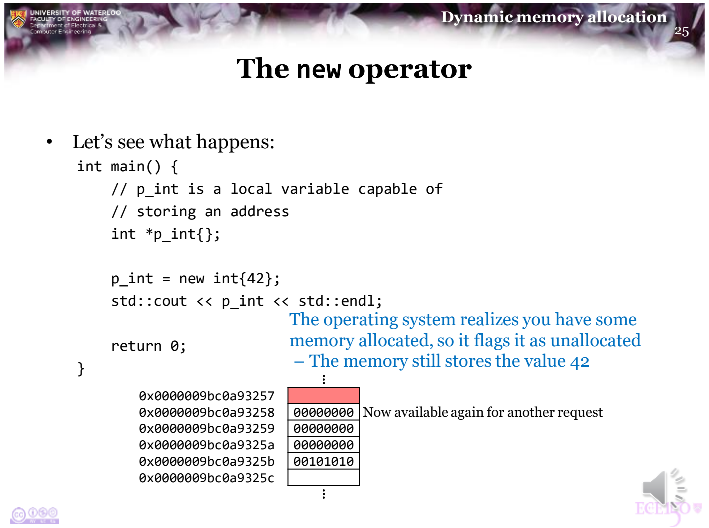
The new operator

- Let's see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

Your program exits

```
                          ⋮
0x0000009bc0a93257
0x0000009bc0a93258   00000000  Allocated to your executing program
0x0000009bc0a93259   00000000
0x0000009bc0a9325a   00000000
0x0000009bc0a9325b   00101010
0x0000009bc0a9325c
                          ⋮
```



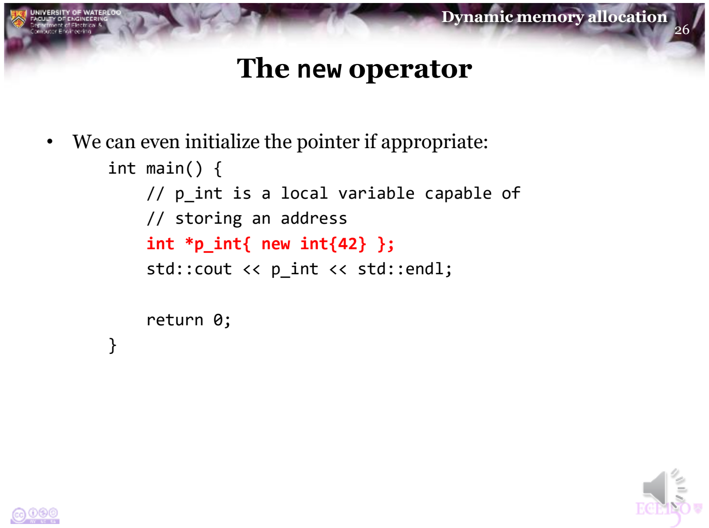
The new operator

- Let’s see what happens:

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    return 0;
}
```

The operating system realizes you have some memory allocated, so it flags it as unallocated
– The memory still stores the value 42

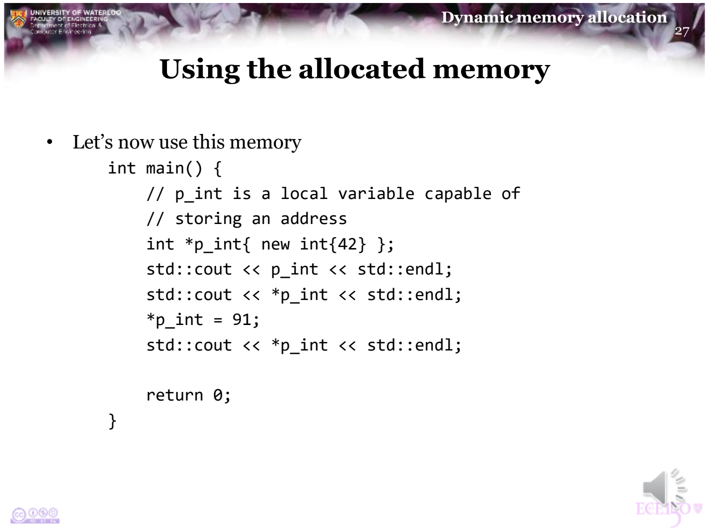
	⋮	
0x0000009bc0a93257		
0x0000009bc0a93258	00000000	Now available again for another request
0x0000009bc0a93259	00000000	
0x0000009bc0a9325a	00000000	
0x0000009bc0a9325b	00101010	
0x0000009bc0a9325c		
	⋮	



The new operator

- We can even initialize the pointer if appropriate:

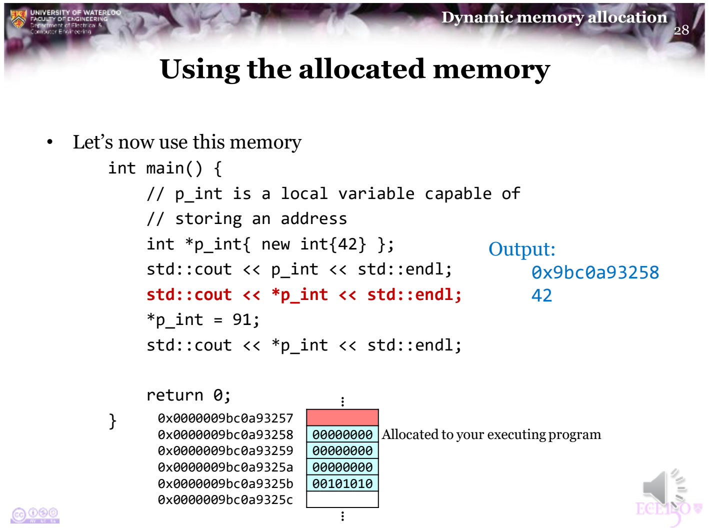
```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{ new int{42} };
    std::cout << p_int << std::endl;

    return 0;
}
```



Using the allocated memory

- Let’s now use this memory

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{ new int{42} };
    std::cout << p_int << std::endl;
    std::cout << *p_int << std::endl;
    *p_int = 91;
    std::cout << *p_int << std::endl;

    return 0;
}
```



Using the allocated memory

- Let’s now use this memory

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{ new int{42} };
    std::cout << p_int << std::endl;
    std::cout << *p_int << std::endl;
    *p_int = 91;
    std::cout << *p_int << std::endl;

    return 0;
}
```

Output:

```
0x9bc0a93258
42
```

	⋮	
0x0000009bc0a93257		
0x0000009bc0a93258	00000000	Allocated to your executing program
0x0000009bc0a93259	00000000	
0x0000009bc0a9325a	00000000	
0x0000009bc0a9325b	00101010	
0x0000009bc0a9325c		
	⋮	

Using the allocated memory

- Let's now use this memory

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{ new int{42} };
    std::cout << p_int << std::endl;
    std::cout << *p_int << std::endl;
    *p_int = 91;
    std::cout << *p_int << std::endl;

    return 0;
}
```

Output:

```
0x9bc0a93258
42
```

```
                         ⋮
0x0000009bc0a93257  |        |
0x0000009bc0a93258  |00000000|  Allocated to your executing program
0x0000009bc0a93259  |00000000|
0x0000009bc0a9325a  |00000000|
0x0000009bc0a9325b  |01011011|
0x0000009bc0a9325c  |        |
                         ⋮
```

Using the allocated memory

- Let's now use this memory

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{ new int{42} };
    std::cout << p_int << std::endl;
    std::cout << *p_int << std::endl;
    *p_int = 91;
    std::cout << *p_int << std::endl;

    return 0;
}
```

Output:

```
0x9bc0a93258
42
91
```

```
                         ⋮
0x0000009bc0a93257  |        |
0x0000009bc0a93258  |00000000|  Allocated to your executing program
0x0000009bc0a93259  |00000000|
0x0000009bc0a9325a  |00000000|
0x0000009bc0a9325b  |01011011|
0x0000009bc0a9325c  |        |
                         ⋮
```

The new operator

- Note that the operating system cleans up your mess after your program exits
  - Is this a good idea?
  - Suppose you open a tab on your web browser
    - That tab requires memory to be dynamically allocated
      - A lot of memory if it is, for example, YouTube

  - Suppose you now close that tab…
    - Is it necessary that that memory remain allocated to the browser?

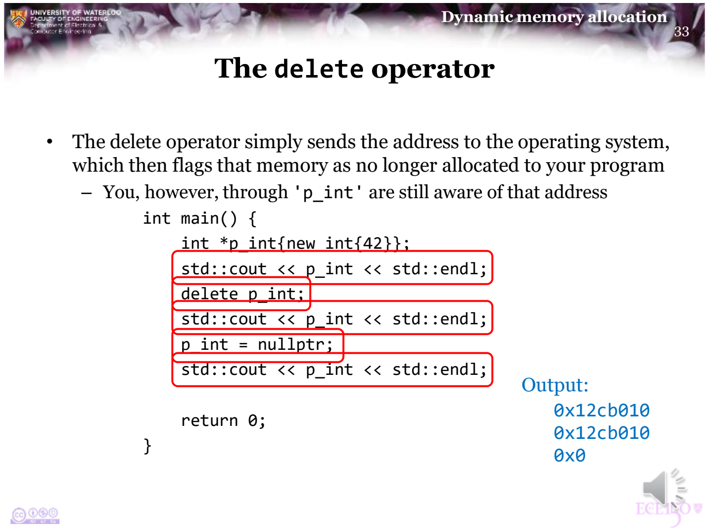
The delete operator

- Just like programs can request memory, programs are able to explicitly tell the operating system when that memory is no longer needed

```
int main() {
    // p_int is a local variable capable of
    // storing an address
    int *p_int{};

    p_int = new int{42};
    std::cout << p_int << std::endl;

    delete p_int;
    p_int = nullptr;

    return 0;
}
```

The address stored in p_int is sent to the operating system

Next, we want to forget this address, so we set p_int to the zero address



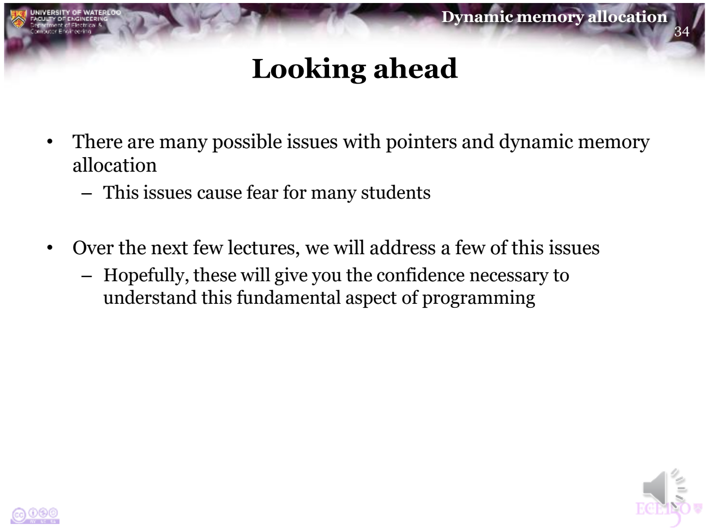
The `delete` operator

- The delete operator simply sends the address to the operating system, which then flags that memory as no longer allocated to your program
  - You, however, through `'p_int'` are still aware of that address

```
int main() {
    int *p_int{new int{42}};
    std::cout << p_int << std::endl;
    delete p_int;
    std::cout << p_int << std::endl;
    p_int = nullptr;
    std::cout << p_int << std::endl;

    return 0;
}
```

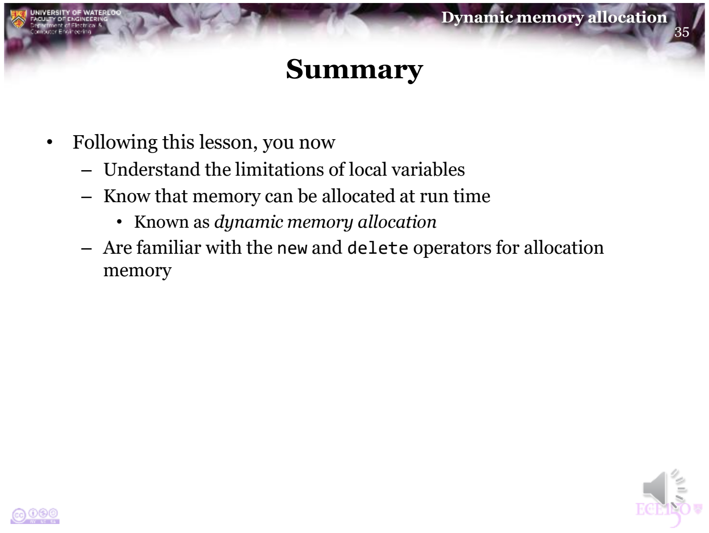
Output:

```
0x12cb010
0x12cb010
0x0
```



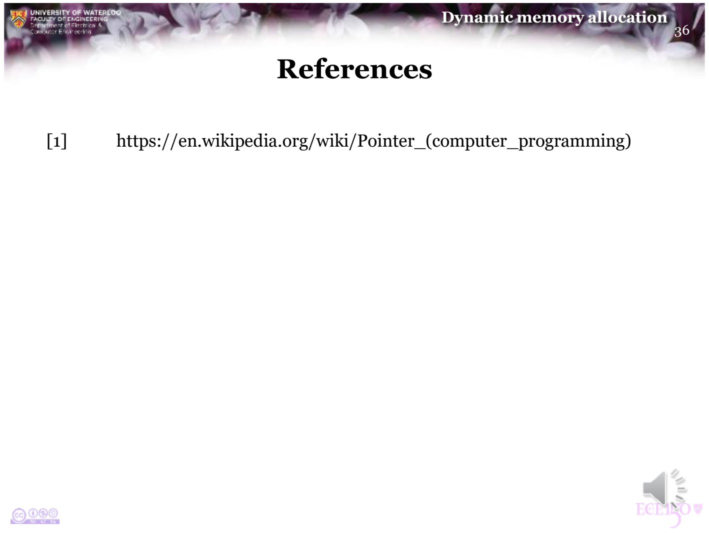
Looking ahead

- There are many possible issues with pointers and dynamic memory allocation
  - This issues cause fear for many students
- Over the next few lectures, we will address a few of this issues
  - Hopefully, these will give you the confidence necessary to understand this fundamental aspect of programming



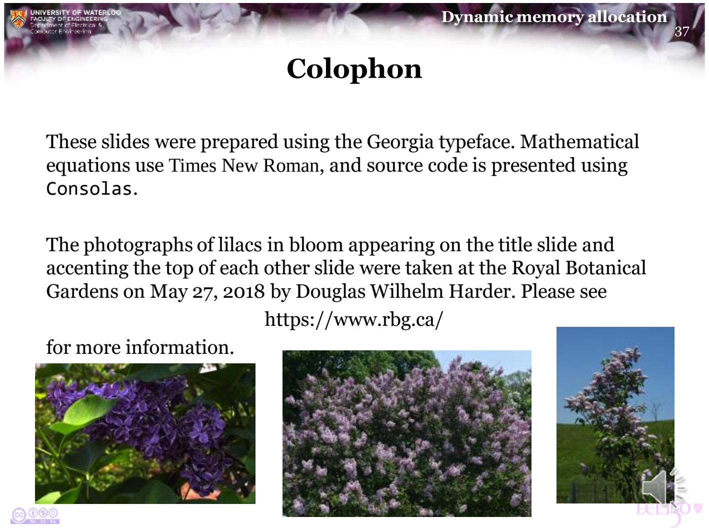
Summary

- Following this lesson, you now
  - Understand the limitations of local variables
  - Know that memory can be allocated at run time
    - Known as *dynamic memory allocation*
  - Are familiar with the `new` and `delete` operators for allocation memory



References

[1] https://en.wikipedia.org/wiki/Pointer_(computer_programming)



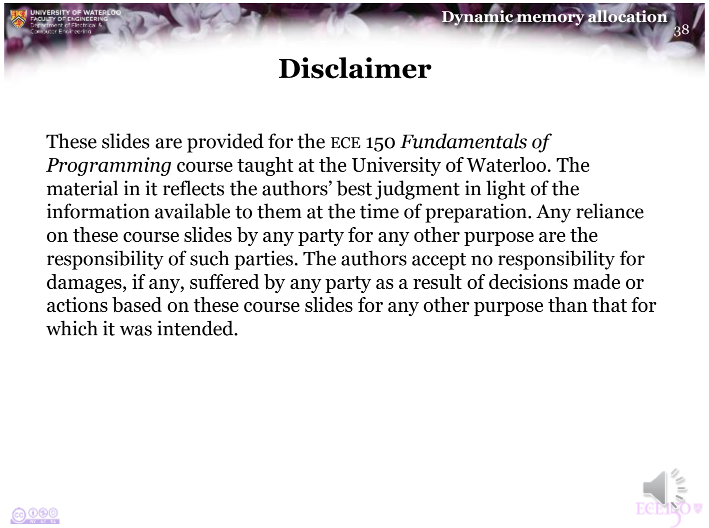
Colophon

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.