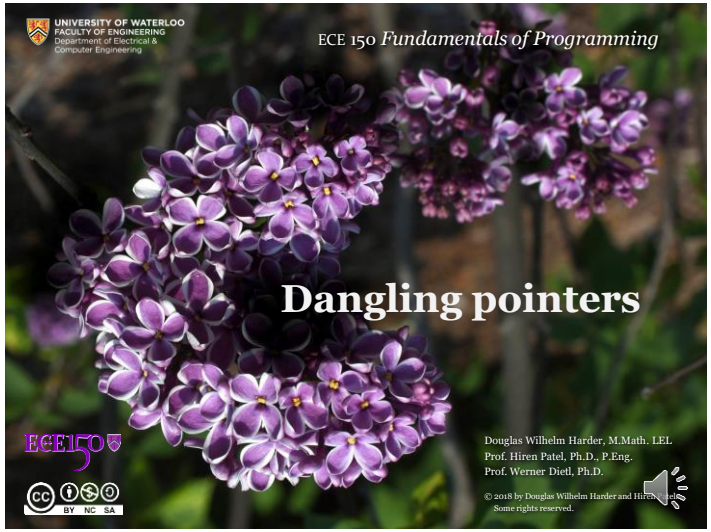

Outline

- In this lesson, we will:
  - Review what happens when we call `delete`
  - See that
    - These pointers still hold their values
    - You can still access and manipulate this deallocated memory
  - This can lead to very difficult-to-find bugs
  - The solution is straight-forward, but requires rigorous control on values

Dangling pointers

- A pointer has a value
  - If the value is `'nullptr'`, this is a known invalid address
  - If initialized or assigned a value returned by `new`, the pointer stores a valid address
  - Once memory is deallocated, it is no longer valid memory
  - A *dangling pointer* is a pointer that stores an address that is no longer allocated

Dangling pointers

- Remember that all `delete` does is pass the value of the operand to the operating system

```
#include <iostream>

int main();

int main() {
    int *p_value{new int{42}};
    std::cout << "Before delete: " << p_value  << std::endl;
    delete p_value;
    std::cout << "After delete:  " << p_value  << std::endl;

    return 0;
}
```

Output:

```
Before delete: 0x1588010
After delete:  0x1588010
```

Dangling pointers

- Because a dangling pointer still stores an address, it is similar to a wild pointer, only the behavior is much worse
  - A wild pointer is almost certainly invalid
  - A dangling pointer stores an address that has previously been allocated
  - Operating systems may flag deallocated memory as available, but may not waste the time to flag it as no longer available
    - This is reasonable:
      - If the same program asks for more memory, if possible it will use this memory first
      - If another program asks for memory and no more memory is available, it may then reclaim deallocated memory assigned to another program

Dangling pointers

- Two separate requests for memory will result in two different addresses

```
#include <iostream>
int main();

int main() {
    int *p_value_1{new int{42}};
    int *p_value_2{new int{91}};

    std::cout << "First new:  " << p_value_1  << std::endl;
    std::cout << "Second new: " << p_value_2  << std::endl;

    delete p_value_1;
    delete p_value_2;

    return 0;
}
```

Output:

```
First new:  0x1ea5010
Second new: 0x1ea5030
```

Dangling pointers

- If memory has been deallocated, the operating system is welcome to reuse it for future requests

```
#include <iostream>
int main();

int main() {
    int *p_value_1{new int{42}};
    std::cout << "First new:  " << p_value_1  << std::endl;
    delete p_value_1;

    int *p_value_2{new int{91}};
    std::cout << "Second new: " << p_value_2  << std::endl;
    delete p_value_2;

    return 0;
}
```

Output:

```
First new:  0x1ea5010
Second new: 0x1ea5010
```

Dangling pointers

- Now, the same memory is being used for two purposes

```
#include <iostream>
int main();

int main() {
    long *p_value_1{ new long{42} };
    delete p_value_1;

    double *p_value_2{ new double{91.0} };
    std::cout << "Before: " << *p_value_2  << std::endl;

    *p_value_1 = 150;
    std::cout << "After:  " << *p_value_1  << std::endl;
    std::cout << "After:  " << *p_value_2  << std::endl;

    return 0;
}
```

Output:

```
Before: 91
After:  150
After:  7.41098e-322
```

Dangling pointers

- A thought experiment:
  - Imagine what may happen if the two pointers, one dangling off of the other, use the same memory for similar purposes…

Dangling pointers

- However, once in a blue moon, the operating system may actually take that deallocated memory away and give it to another program

```
#include <iostream>
int main();

int main() {
    int *p_value{new int{42}};
    delete p_value;

    for ( int k{0}; k < 1000000000; ++k ) {
        // Do nothing...
    }

    *p_value = 91;
    std::cout << *p_value << std::endl;
    return 0;
}
```

Output after 57368 executions:

```
Segmentation fault (core dumped)
```


Dangling pointers

- This can be solved by always assigning deallocated pointers the value of `nullptr`:

```
#include <iostream>
int main();

int main() {
    int *p_value{new int{42}};
    // Use the value...

    delete p_value;
    p_value = nullptr;

    return 0;
}
```

  - You should always assign deallocated pointers the value `nullptr` even if the function is exiting—a wild pointer may pick up that value

Dangling pointers

- This becomes more difficult if multiple pointers are assigned the same address:

```
#include <iostream>
int main();

int main() {
    int *p_value_1{new int{42}};
    std::cout << "Address 1: " << p_value_1 << std::endl;
    int *p_value_2{p_value_1};

    delete p_value_1;
    p_value_1 = nullptr;
    std::cout << "Address 2: " << p_value_2 << std::endl;
    *p_value_2 = 91;
    std::cout << "New value: " << *p_value_2 << std::endl;

    return 0;
}
```

Output:

```
Address 1: 0x1dc9010
Address 2: 0x1dc9010
New value: 91
```

Warning!

- The single most greatest difficult for students studying algorithms and data structures is this *belief* that delete does something magical
  - The statement

    ```
    delete p_data;
    ```

    does **nothing** more than send the address to the operating system
- The operating system simply flags the memory at that location as being available for future allocations

Summary

- Following this lesson, you now
  - Understand that `delete` does not change the value of a pointer
  - Know that calling delete on a pointer:
    - Allows the operating system to flag the memory as available
    - The operating system, however, usually still leaves the memory allocated to the task…usually
  - When a pointer is deallocated, it must be assigned `nullptr`
  - Otherwise, `delete` does nothing to change the value of the pointer

References

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

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