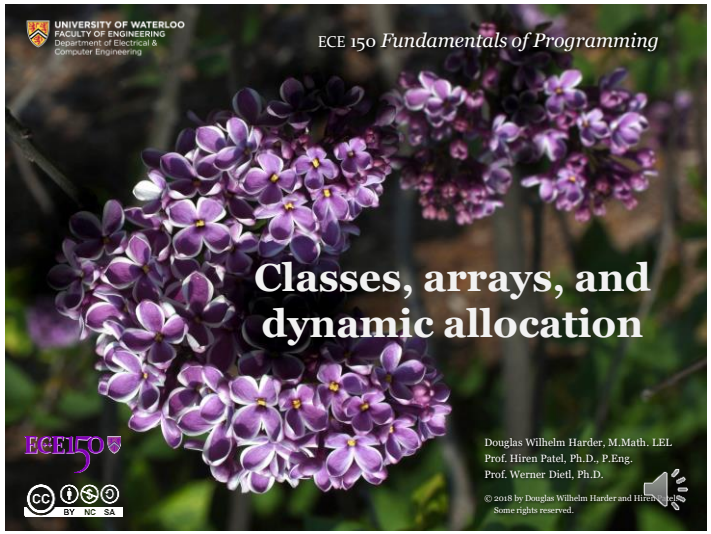


1

Outline

- In this lesson, we will:
  - Review how arrays of int are initialized
  - Look at how to initialize an array of objects
  - Look at when the constructor and destructor are called
  - See how to dynamically allocate an instance of a class
    - Understanding when the constructor and destructor are called
  - See how to dynamically allocate an array of instances of a class
  - Learn to call member functions and access member variables on pointers to objects with the -> operator
  - Learn some basic subtleties of pointers and objects

2

Local arrays

- Recall the behavior of initializing local arrays:

```
// Set all values to the default
int data[5]{};

// Set the entries to 17, 35, 0, 0, 0
//  - after the first two, the rest are set to the default
int data[5]{17, 35};

// Sets the five entries to 11, 12, 13, 14, 15
int data[5]{11, 12, 13, 14, 15};
```

3

Local arrays

- Consider this class:

```
class C {
    public:
        C(double x = 0.0);
        C(int m, int n = 0);
        std::string about() const;
};

C::C(double x) {
    std::cout << "Calling C(double)" << std::endl;
}

C::C(int m, int n) {
    std::cout << "Calling C(int, int)" << std::endl;
}
std::string C::about() const {
    return "Harmless";
}
```

4

Local arrays

- Creating an array of this class is as follows:

```
int main() {
    C data[10]{ {1}, {}, {5, 7}, {2.5} };
    return 0;
}
```

Output:

```
Calling C(int, int)
Calling C(double)
Calling C(int, int)
Calling C(double)
Calling C(double)
Calling C(double)
Calling C(double)
Calling C(double)
Calling C(double)
Calling C(double)
```

5

Local arrays

- We can now call member functions on these entries:

```
int main() {
    C data[10]{ {1}, {}, {5, 7}, {2.5} };

    for ( std::size_t k{0}; k < 10; ++k ) {
        std::cout << data[k].about() << std::endl;
    }

    return 0;
}
```

Continued output:

```
Harmless
Harmless
Harmless
Harmless
Harmless
Harmless
Harmless
Harmless
Harmless
Harmless
```

6

Local arrays

- If a class does not have a constructor taking no arguments, you must provide arguments for all entries in the local array

```
class D {
    public:
        D(int n);
        std::string about() const;
};

D::D( int n ) {
   std::cout << "Calling D(int)" << std::endl;
}

std::string D::about() const {
   return "Mostly harmless";
}
```

7

Local arrays

- Now sufficient initial values must be given to fill up the array

```
int main() {
    D data[5]{ {1}, {2}, {3}, {4}, {5} };
    return 0;

    for ( std::size_t k{0}; k < 5; ++k ) {
        std::cout << data[k].about() << std::endl;
    }

    return 0;
}
```

Output:

```
Calling D(int)
Calling D(int)
Calling D(int)
Calling D(int)
Calling D(int)
Mostly harmless
Mostly harmless
Mostly harmless
Mostly harmless
Mostly harmless
```

8

Local arrays

- When a local array goes out of scope,
  the destructor is called on each entry of that array

```
class E {
    public:
        ~E();
};

E::~E() {
    std::cout << "Calling E~()" << std::endl;
}

int main() {
    E data[5]{};
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Output:

```
Hello world!
Calling ~E()
Calling ~E()
Calling ~E()
Calling ~E()
Calling ~E()
```

Objects as arguments to functions

- If a parameter is declared to be passed by value,
  when that function is called, the copy constructor is called to initialize that parameter as a copy of the argument
  - Additionally, when the function returns,
    the destructor is called on that parameter
- If a parameter is declared to be passed by reference,
  when that function is called,
  the parameter is an alias for the argument
  - When the function returns, no destructor is called,
    only the reference variable goes out of scope

Objects as arguments to functions

- For example,

```
class F {
    public:
        F();
        F( F const &original );
       ~F();
};

F::F() {
    std::cout << " - Calling F()" << std::endl;
}

F::F( F const &original ) {
    std::cout << " - Calling F(F const &)" << std::endl;
}

F::~F() {
    std::cout << " - Calling F~()" << std::endl;
}
```

Objects as arguments to functions

- For example,

```
int main() {
    F obj{};

    std::cout << "Passing by reference:" << std::endl;
    by_reference( obj );
    std::cout << "Passing by value:" << std::endl;
    by_value( obj );
    std::cout << "Returning from main:" << std::endl;

    return 0;
}

void by_reference( F &ref_param ) {
}

void by_value( F param ) {
}
```

Output:

```
 - Calling F()
Passing by reference:
Passing by value:
 - Calling F(F const &)
 - Calling F~()
Returning from main:
 - Calling F~()
```

Array of objects passed to a function

- Suppose you have a function that takes an array of objects

```
void by_array( F array[], std::size_t const capacity );
```

  - This is just like any other array: just the address is passed
  - The address passed is the address of the original array, so no constructor or destructor need be called


Array of objects passed to a function

- Consider this:

```
class G {
    public:
        G( int n );
        G( G const &original );
       ~G();
        int retrieve() const;
    private:
        int value_;
};
```

```
G::G( int new_value ):
value_{ new_value } {
    std::cout << "Calling G(int)" <<
std::endl;
}

G::G( G const &original ):
value_{ original.value_ } {
    std::cout << "Calling G(G const &)"
              << std::endl;
}

G::~G() {
    std::cout << "Calling ~G()"
              << std::endl;
}

int G::retrieve() const {
    return value_;
}
```


Array of objects passed to a function

- Suppose you have a function

```
int main() {
    G data[5]{ {10}, {11}, {12}, {13}, {14} };

    print( data, 5 );
    return 0;
}

void print( G array[], std::size_t capacity ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        std::cout << array[k].retrieve() << std::endl;
    }
}
```

Output:

```
Calling G(int)
Calling G(int)
Calling G(int)
Calling G(int)
Calling G(int)
10
11
12
13
14
Calling ~G()
Calling ~G()
Calling ~G()
Calling ~G()
Calling ~G()
```

Dynamically allocated memory

- What we haven't discussed yet is the dynamic allocation of objects
  - It works just like should expect:
    - You could call `new int{}` or `new int{42}`
    - When you call `new Class_name{…}` for a single instance, you can pass the arguments for the initialization
    - The compiler decides which constructor you meant to call
  - If there area no constructors that take zero arguments, you must pass the minimum required arguments
  - When you call `delete`, the destructor is called
    - If you forget to call `delete`, the destructor is never called

Dynamically allocated memory

- Using our last class G,

```
int main() {
    G *p_item{ new G{3} };

    delete p_item;
    p_item = nullptr;

    return 0;
}
```

```
class G {
    public:
        G( int n );
        G( G const &original );
       ~G();

        int retrieve() const;
    private:
        int value_;
};
```

Output:

```
Calling G(int)
Calling ~G()
```

Dynamically allocated memory

- Using our last class G,

```
int main() {
    G *a_items{ new G[3]{ {101}, {102}, {103} } };

    delete[] a_items;
    a_items = nullptr;

    return 0;
}
```

Output:

```
Calling G(int)
Calling G(int)
Calling G(int)
Calling ~G()
Calling ~G()
Calling ~G()
```

Warning!!!!!

- Using our last class G,

```
int main() {
    G *a_items{ new G[3]{ {101}, {102}, {103} } };

    delete a_items;
    a_items = nullptr;

    return 0;
}
```

Output:

```
Calling G(int)
Calling G(int)
Calling G(int)
Calling ~G()
```

Accessing member variables and functions

- To access a member variable or function on a pointer to an instance of a class, you use -> and not .

```
int main() {
    G item{ 42 };
    G *p_item{ new G{91} };

    std::cout <<   item.retrieve() << std::endl;
    std::cout << p_item->retrieve() << std::endl;

    delete p_item;
    p_item = nullptr;

    return 0;
}
```

```
class G {
    public:
        G( int n );
        G( G const &original );
       ~G();

        int retrieve() const;
    private:
        int value_;
};
```

Output:

```
Calling G(int)
Calling G(int)
42
91
Calling ~G()
Calling ~G()
```

Dynamically allocated memory

- Recall that with a dynamically allocated array of integers, the indexing operator already gives us the k^{th} entry

```
int main() {
    int *a_items{ new int[3]{ 101, 102, 103 } };

    for ( std::size_t k{0}; k < 3; ++k ) {
        std::cout << a_items[k] << std::endl;
    }

    delete[] a_items;
    a_items = nullptr;

    return 0;
}
```

Output:

```
101
102
103
```

21

Dynamically allocated memory

- Similarly, if we have a dynamically allocated array of objects, the index accesses the entry of the array, so we use the dot operator .

```
int main() {
    G *a_items{ new G[3]{ {101}, {102}, {103} } };

    for ( std::size_t k{0}; k < 3; ++k ) {
        std::cout << a_items[k].retrieve() << std::endl;
    }

    delete[] a_items;
    a_items = nullptr;

    return 0;
}
```

Output:

```
Calling G(int)
Calling G(int)
Calling G(int)
101
102
103
Calling ~G()
Calling ~G()
Calling ~G()
```

22

Accessing member variables and functions

- You can have an array of pointers:

```
int main() {
    G *array[5]{};

    for ( std::size_t k{0}; k < 5; k += 2 ) {
        array[k] = new G{ 100 + k };
    }

    for ( std::size_t k{0}; k < 5; ++k ) {
        if ( array[k] != nullptr ) {
            std::cout << array[k]->retrieve()
                      << std::endl;
            delete array[k];
            array[k] = nullptr;
        }
    }

    return 0;
}
```

Output:

```
Calling G()
Calling G()
Calling G()
100
Calling ~G()
102
Calling ~G()
104
Calling ~G()
```

23

Remembering which...

- Rule:
  - If you can print the variable (a local variable or parameter, a reference to either of these) or array entry, and printing it produces an address, use the arrow operator ->
  - Otherwise, use the dot operator .
    - Note, if you have not overloaded the appropriate `operator<<` for printing an instance of a particular class, printing it may cause a compilation error
    - You can always print addresses

24

Remembering which…

- Guideline:
  - If you meticulously follow the naming convention of prefixing pointers with `p_name`,
    this will help you remember when to use `p_name->member(…)`
    - You will call `delete p_name;` on any such variable
  - If you prefix any dynamically allocated array with `a_name`,
    this will help you to remember to use `a_name[k].member(…)`
    - You will call `delete[] a_name;` on any such variable
  - Otherwise, if you have just *name* use the dot operator *name*`.member(…)`
- It is possible to have pointers to pointers to …, etc.,
  but that is currently beyond the scope of this class ☺

One last example

- Suppose we have the following class:

```
class Node {
    public:
        int value_;
        Node *p_next_;
};
```

- This class stores:
  - An integer
  - An address of another instance of this class
    - That is, a pointer to another instance of this class

One last example

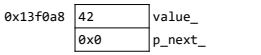
- Let's use this class:

```
int main() {
    Node *p_42{ new Node{ 42, nullptr } };
    std::cout << " p_42 == " << p_42 << std::endl;
    Node *p_91{ new Node{ 91, p_42 } };
    std::cout << " p_42 == " << p_42 << std::endl;

    std::cout << p_91->value_    << std::endl;
    std::cout << p_91->p_next_   << std::endl;
    std::cout << p_91->p_next_->value_ << std::endl;
    std::cout << p_91->p_next_->p_next_ << std::endl;

    delete p_91->p_next_;
    p_91->p_next_ = nullptr;
    // We can no longer call delete p_42;
    p_42 = nullptr; // Dangling pointer!!!
    delete p_91;
    p_91 = nullptr;

    return 0;
```



Output:

```
p_42 == 0x13f0a8
p_91 == 0xb03c50
91
0x13f0a8
42
0
```

```
class Node {
    public:
        int value_;
        Node *p_next_;
};
```


Summary

- Following this lesson, you now
  - Know how to initialize arrays of objects
    - Initial values are required if no constructor takes no values
  - Understand when the constructor and destructor is called
  - Know how to dynamically allocate and delete objects and arrays of objects
  - Have a better understanding of arrays and pointers
  - Have been introduced to using the . operator and the -> operator
  - Have been introduced to classes that have member variables that are themselves pointers

References

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

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.