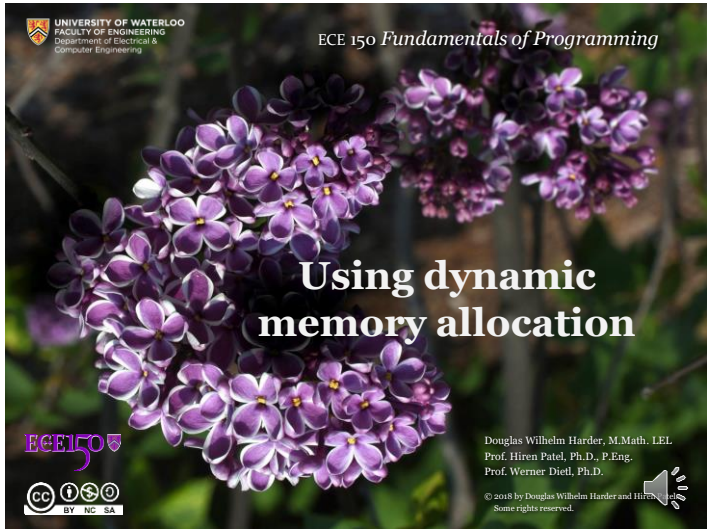


Outline

- In this lesson, we will:
  - Discuss the lifetime of dynamically allocated memory
  - Author and discuss a program that requires repeated dynamic memory allocation and deallocation
  - Be responding to such demands as a result of external need

The problem

- We will solve the following problem:
  - Suppose we are reading data
    - We will read it from the console, but it could be from a sensor
  - We do not know *a priori* how much data there will be:
    - There could be only one datum, there could be a thousand, or one million
  - We don't want to waste too much space
    - No point in allocating an array of capacity one million if there are only going to be 7 data points…

Helper function

- Here is our helper function, to print out an array:

```
void print_array( double *array,
                  std::size_t const capacity ) {
    if ( capacity == 0 ) {
        return;
    }

    std::cout << array[0];

    for ( std::size_t k{1}; k < capacity; ++k ) {
        std::cout << ", " << array[k];
    }

    std::cout << std::endl;
}
```

Basic framework

```
int main() {
    std::size_t data_capacity{ ?? };
    double *data{ new double[data_capacity] };

    while ( true ) {
        double x{};
        std::cout << "Enter a number (<= 0 to quit): ";
        std::cin >> x;

        if ( x <= 0.0 ) {
            break;
        }

        // Store the new datum 'x'
    }

    print_array( data, data_size );
    delete[] data;
    data = nullptr;

    return 0;
}
```


Initial set up

- Let us start with an array of capacity 10:

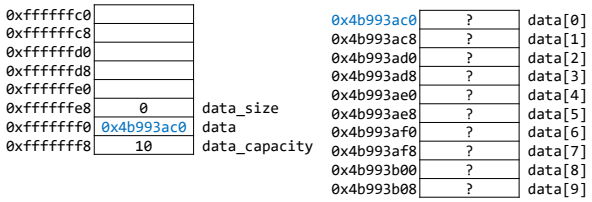
```
std::size_t data_capacity{ 10 };
double *data{ new double[data_capacity] };
```

- While the capacity is 10,
  we have not yet entered any data into this array
  - We need a second variable storing how many values have been stored in the array

```
std::size_t data_size{ 10 };
```


Initial set up

- Thus, our initial set-up looks like the following:

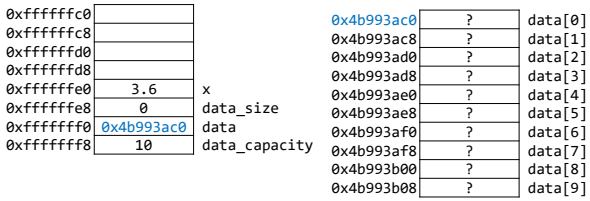


Initial set up

```
int main() {
    std::size_t data_capacity{ 10 };
    double *data{ new double[data_capacity] };
    std::size_t data_size{0};

    while ( true ) {
        double x{};
        std::cout << "Enter a number (<= 0 to quit): ";
        std::cin >> x;

        if ( x <= 0.0 ) {
            break;
        }

        // Store the new datum 'x'
    }

    print_array( data, data_size );
    delete[] data;
    data = nullptr;

    return 0;
}
```

Storing data

- When the first datum x arrives, we must put it into `data[0]`
  - Following this, the *size* is now 1



  - Fortunately, `data_size` tells us where to put the next value:
    - If there are `data_size` entries filled up, the next must be placed at `data[data_size]`
    - Then we increment `data_size`

Storing data

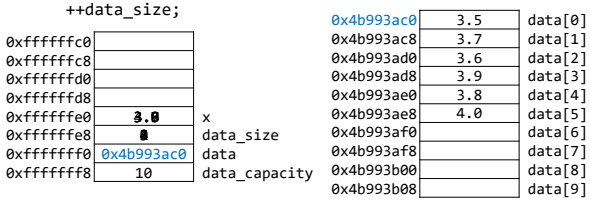
- Let us focus on this loop:

```
while ( true ) {
    double x{};
    std::cout << "Enter a number (<= 0 to quit): ";
    std::cin >> x;

    if ( x <= 0.0 ) {
        break;
    }

    data[data_size] = x;
    ++data_size;
}
```


Storing data

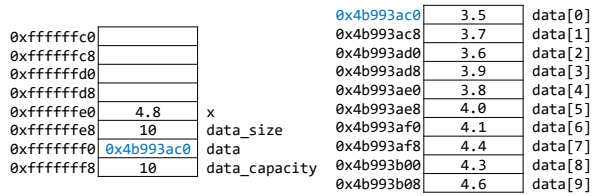
- Watching the first few steps:

```
data[data_size] = x;
++data_size;
```



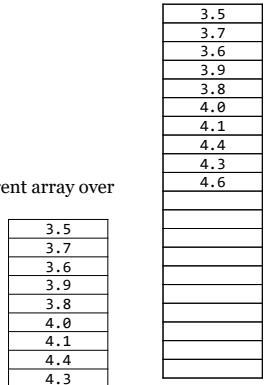
Storing data

- After ten steps, the array is full



Dealing with a full array

- What can we do if the array is full?
  - We cannot expand the array
- Strategy:
  - We could request a larger array, and copy anything in the current array over



Dealing with a full array

- Again, focusing on the loop:

```
while ( true ) {
    double x{};
    std::cout << "Enter a number (<= 0 to quit): ";
    std::cin >> x;

    if ( x <= 0.0 ) {
        break;
    }

    if ( data_size == data_capacity ) {
        // We are full...increase the array capacity
    }

    data[data_size] = x;
    ++data_size;
}
```

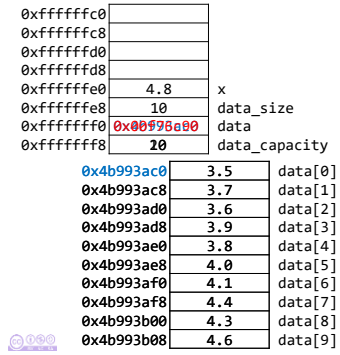

Dealing with a full array

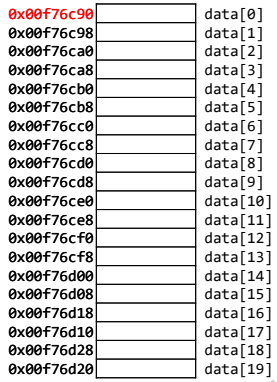
- We could either:
  - Double our array capacity
  - Increase the array capacity by 10
- Let's choose the first:

```
if ( data_size == data_capacity ) {
    data_capacity *= 2;
    data = new double[data_capacity];
    // Copy everything over...
}
```


Dealing with a full array

```
data_capacity *= 2;
data = new double[data_capacity];
```





Dealing with a full array

- Can we store the old information first?

```
if ( data_size == data_capacity ) {
    std::size_t old_capacity{ data_capacity };
    double *old_data{ data };
    data_capacity *= 2;
    data = new double[data_capacity];
    // Copy everything over...
}
```


Dealing with a full array

```
std::size_t old_capacity{ data_capacity };
double *old_data{ data };
data_capacity *= 2;
data = new double[data_capacity];
```

0xffffffc0		
0xffffffc8		
0xffffffd0	0x4b993ac0	old_data
0xffffffd8	10	old_capacity
0xffffffe0	4.8	x
0xffffffe8	10	data_size
0xfffffff0	0x00f76a00	data
0xfffffff8	20	data_capacity

0x4b993ac0	3.5	old_data[0]
0x4b993ac8	3.7	old_data[1]
0x4b993ad0	3.6	old_data[2]
0x4b993ad8	3.9	old_data[3]
0x4b993ae0	3.8	old_data[4]
0x4b993ae8	4.0	old_data[5]
0x4b993af0	4.1	old_data[6]
0x4b993af8	4.4	old_data[7]
0x4b993b00	4.3	old_data[8]
0x4b993b08	4.6	old_data[9]

0x00f76c90		data[0]
0x00f76c98		data[1]
0x00f76ca0		data[2]
0x00f76ca8		data[3]
0x00f76cb0		data[4]
0x00f76cb8		data[5]
0x00f76cc0		data[6]
0x00f76cc8		data[7]
0x00f76cd0		data[8]
0x00f76cd8		data[9]
0x00f76ce0		data[10]
0x00f76ce8		data[11]
0x00f76cf0		data[12]
0x00f76cf8		data[13]
0x00f76d00		data[14]
0x00f76d08		data[15]
0x00f76d18		data[16]
0x00f76d10		data[17]
0x00f76d28		data[18]
0x00f76d20		data[19]

Dealing with a full array

- Now we have to copy the old information over:

```
if ( data_size == data_capacity ) {
    std::size_t old_capacity{ data_capacity };
    double *old_data{ data };
    data_capacity *= 2;
    data = new double[data_capacity];

    for ( std::size_t k{0}; k < old_capacity; ++k ) {
        data[k] = old_data[k];
    }
}
```


Dealing with a full array

```
if ( data_size == data_capacity ) {
    // Store old data, allocate new array...

    for ( std::size_t k{0}; k < old_capacity; ++k ) {
        data[k] = old_data[k];
    }
}
```

0xffffffc0		
0xffffffc8	0	k
0xffffffd0	0x4b993ac0	old_data
0xffffffd8	10	old_capacity
0xffffffe0	4.8	x
0xffffffe8	10	data_size
0xfffffff0	0x00f76c90	data
0xfffffff8	20	data_capacity

0x4b993ac0	3.5	old_data[0]
0x4b993ac8	3.7	old_data[1]
0x4b993ad0	3.6	old_data[2]
0x4b993ad8	3.9	old_data[3]
0x4b993ae0	3.8	old_data[4]
0x4b993ae8	4.0	old_data[5]
0x4b993af0	4.1	old_data[6]
0x4b993af8	4.4	old_data[7]
0x4b993b00	4.3	old_data[8]
0x4b993b08	4.6	old_data[9]

0x00f76c90	3.5	data[0]
0x00f76c98	3.7	data[1]
0x00f76ca0	3.6	data[2]
0x00f76ca8	3.9	data[3]
0x00f76cb0	3.8	data[4]
0x00f76cb8	4.0	data[5]
0x00f76cc0	4.1	data[6]
0x00f76cc8	4.4	data[7]
0x00f76cd0	4.3	data[8]
0x00f76cd8	4.6	data[9]
0x00f76ce0	4.8	data[10]
0x00f76ce8		data[11]
0x00f76cf0		data[12]
0x00f76cf8		data[13]
0x00f76d00		data[14]
0x00f76d08		data[15]
0x00f76d18		data[16]
0x00f76d10		data[17]
0x00f76d28		data[18]
0x00f76d20		data[19]

Dealing with a full array

- The old array is no longer required

```
if ( data_size == data_capacity ) {
    std::size_t old_capacity{ data_capacity };
    double *old_data{ data };
    data_capacity *= 2;
    data = new double[data_capacity];

    for ( std::size_t k{0}; k < old_capacity; ++k ) {
        data[k] = old_data[k];
    }

    delete[] old_array;
    old_array = nullptr;
}
```


Allocating instances of a type

```
while ( true ) {
    double x{};
    std::cout << "Enter a number (<= 0 to quit): ";
    std::cin >> x;

    if ( x <= 0.0 ) {
        break;
    }

   if ( data_size == data_capacity ) {
       std::size_t old_capacity{ data_capacity };
       double *old_data{ data };
       data_capacity *= 2;
       data = new double[data_capacity];

       for ( std::size_t k{0}; k < old_capacity; ++k ) {
           data[k] = old_data[k];
       }

       delete[] old_array;
       old_array = nullptr;
   }

    data[data_size] = x;
    ++data_size;
}
```


Summary

- Following this lesson, you now
  - Understand the process of dynamic memory allocation
  - Know we must keep track of all previously allocated memory until it is no longer required
    - At which point it can be deallocated
  - Are aware that all addresses must be stored in local variables in order to access them
  - Have reinforced your understanding that once a local variable goes out of scope, any information it contains is now gone

References

[1] https://en.wikipedia.org/wiki/New_and_delete_(C++)

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.