Using dynamic memory allocation

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:
  ```c++
  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;
  }
  ```
Using dynamic memory allocation

**Basic framework**

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

    while ( true ) {
        double x{0};
        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:
  ```cpp
  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
    ```cpp
    std::size_t data_size{0};
    ```

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffffc0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffc8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffd0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffd8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffe0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xffffffffe8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xfffffffff0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0xfffffffff8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993ac0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993ac8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993ad0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993ad8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993ae0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993ae8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993af0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993af8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993bf0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb993bf8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
```
Using dynamic memory allocation

Storing data

• When the first datum $x$ arrives, we must put it into $\text{data}[0]$
  
  Following this, the size is now 1

| 0xfffffffc0 | 0x4b993ac8 | data[0] |
| 0xfffffffc8 | 0x4b993ac8 | data[1] |
| 0xfffffffd0 | 0x4b993ad8 | data[2] |
| 0xfffffffd8 | 0x4b993ad8 | data[3] |
| 0xfffffffe0 | 3.6        | x        |
| 0xfffffffe8 | 0      | data_size |
| 0xffffffff0 | 0x4b993ac8 | data_capacity |

• Fortunately, $\text{data\_size}$ tells us where to put the next value:
  
  • If there are $\text{data\_size}$ entries filled up,
    the next must be placed at $\text{data}[\text{data\_size}]$
  
  • Then we increment $\text{data\_size}$

10

Using dynamic memory allocation

Storing data

• Let us focus on this loop:

```cpp
while ( true ) {
    double x{};
    std::cout << "Enter a number (<= 0 to quit): ";
    std::cin >> x;
    if ( x <= 0.0 ) {
        break;
    }
    \text{data}[\text{data\_size}] = x;
    ++\text{data\_size};
}
```

After ten steps, the array is full

| 0xfffffffc0 | 0x4b993ac0 | 3.5 | data[0] |
| 0xfffffffc8 | 0x4b993ac8 | 3.7 | data[1] |
| 0xfffffffd0 | 0x4b993ad8 | 3.6 | data[2] |
| 0xfffffffd8 | 0x4b993ad8 | 3.9 | data[3] |
| 0xfffffffe0 | 3.6        | x   |
| 0xfffffffe8 | 0      | data_size |
| 0xffffffff0 | 0x4b993ac8 | data_capacity |
| 0x4b993ac0 | 0x4b993af8 | data[7] |
| 0x4b993af0 | 0x4b993af8 | data[8] |
| 0x4b993af8 | 0x4b993b00 | data[9] |

10

Watching the first few steps:

```
data[\text{data\_size}] = x;
++\text{data\_size};
```

| 0xfffffffc0 | 0x4b993ac0 | 3.5 | data[0] |
| 0xfffffffc8 | 0x4b993ac8 | 3.7 | data[1] |
| 0xfffffffd0 | 0x4b993ad8 | 3.6 | data[2] |
| 0xfffffffd8 | 0x4b993ad8 | 3.9 | data[3] |
| 0xfffffffe0 | 3.6        | x   |
| 0xfffffffe8 | 0      | data_size |
| 0xffffffff0 | 0x4b993ac8 | data_capacity |
| 0x4b993ac0 | 0x4b993ae8 | 4.0 | data[5] |
| 0x4b993ae0 | 0x4b993af0 | 4.1 | data[6] |
| 0x4b993af8 | 0x4b993af8 | data[7] |
| 0x4b993b00 | 0x4b993b00 | data[8] |
| 0x4b993b08 | 0x4b993b08 | data[9] |

3.5
3.7
3.6
3.9
3.8
4.0
4.1
3.6
3.7
3.8
3.9
4.0
4.1
4.4
4.3
4.6
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

```cpp
while (true) {
    double x;
    cin >> x;
    if (x <= 0.0) {
        break;
    }
    if (data_size == data_capacity) {
        // We are full...increase the array capacity
        data_size = x;
        ++data_size;
    }
}
```

- Again, focusing on the loop:

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

We could either:

- Double our array capacity
- Increase the array capacity by 10

Let's choose the first:

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

- 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

- The old array is no longer required
  
  ```cpp
  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;
  }
  ```

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

Allocating instances of a type

```cpp
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;
}
```

References

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.