Advanced Data Types in C++

Dr Ian Cornelius

Hello

Hello (1)

Learning Outcomes

1. Understand the difference between arrays, vectors and maps that are built-in to C++
2. Demonstrate the ability to create arrays, vectors and maps to store data

Arrays

Arrays (1)

- Almost like a `list` in Python and are used to store multiple items in a single variable
- However, slightly different as they store only a single type of data
  - i.e. double, boolean or integer
- They are considered to be:
  - **ordered**: the items have a defined order, and this order will not change when new items are added to the array
  - **changeable**: the items of an array are mutable (can be changed), added or removed
  - **allowable of duplicates**: arrays are indexed, and therefore items in an array can be duplicated

Arrays (2)

Creating an Array

- Arrays are declared by using a set of square brackets (`[]`) at the end of the variable name
- Inside the set of square brackets will be a number
  - indicates the number of elements that can be stored in the array

```
dataType arrayName[arraySize];
```

- The items inside the array can only be of the data type that was assigned to the variable

```
int intArrayExample1[10];  // Stores only integers
double dblArrayExample1[3]; // Stores only doubles
char charArrayExample1[26];    // Stores only characters
```

Arrays (3)

Initialising an Array

- Array can be populated by using the curly braces (`{}`) after the variable name declaration

```
int intArrayExample1[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
double dblArrayExample1[3] = {1.0, 2.0, 3.0};
char charArrayExample1[26] = {'a', 'b', 'c', 'd', 'e'};
```

Arrays (4)

Accessing an Arrays Element

- The items in an array can be accessed by referring to its index number inside a set of square brackets (`[]`)
    - *remember* that the index of an array begins at `0` in C++

```
int arrayExample11[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
```

```
arrayExample11[0] -> 10
arrayExample11[3] -> 7
arrayExample11[9] -> 1
```

Arrays (5)

Getting the Size of an Array i

- There is no in-built function to obtain the size unlike a Python `list`
- Could use the `sizeof()` function to find the array size

```
int arrayExample11[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
sizeof(arrayExample11);
```

```
sizeof(arrayExample11) -> 40
```

- The above code returns 40 as the size of the array
- The `sizeof()` function returns the size of the data type in bytes
    - as we have 10 integers, and each integer occupies 4 bytes of memory, 40 is returned

Arrays (6)

Getting the Size of an Array ii

- Therefore, if we know the size of the data type, then a simple calculation can return the array size

```
int arrayExample1[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
sizeof(arrayExample1) / sizeof(int);
```

```
sizeof(arrayExample1) / sizeof(int) ≈ 40 / 4
sizeof(arrayExample1) / sizeof(int) -> 10
```

Arrays (7)

Inserting Elements into an Array i

- The elements in an array are **ordered** and **indexed**, therefore modifiable
  - otherwise known as being *mutable*
- Items can be inserted into the array by calling an index number that has not been populated at initial creation

```
int arrayExample1[8] = {10, 9, 8, 7, 6};
arrayExample1[6] = -9;
```

```
arrayExample1[0] -> 10
arrayExample1[1] -> 9
arrayExample1[2] -> 8
arrayExample1[3] -> 7
arrayExample1[4] -> 6
arrayExample1[5] -> 0
arrayExample1[6] -> -9
arrayExample1[7] -> 0
```

Arrays (8)

Inserting Elements into an Array ii

- Elements at a particular index can also be *replaced*
- Achieve by accessing the index of the element you wish to replace

```
int arrayExample1[8] = {10, 9, 8, 7, 6};
arrayExample1[2] = -9;
```

```
[Before] arrayExample1[2] -> 8
[After]  arrayExample1[2] -> -9
```

Arrays (9)

Resizing an Array

- A disadvantage to using an array is the static size construction
  - e.g. `int arrayExample1[10]` can only store ten integers and no more
- A method of resizing an array is to create an array of a new size and copy the contents from the old array

```
int arrayExample1[8] = {10, 9, 8, 7, 6, 5, 4, 3};
int arrayExample2[15];
int main() {
    for(int i = 0; i < (sizeof(arrayExample1) / sizeof(int)); i++) {
        arrayExample2[i] = arrayExample1[i];
    }
    return 0;
}
```

```
arrayExample2[0] -> 10
arrayExample2[1] -> 9
arrayExample2[2] -> 8
arrayExample2[3] -> 7
arrayExample2[4] -> 6
arrayExample2[5] -> 5
arrayExample2[6] -> 4
arrayExample2[7] -> 3
arrayExample2[8] -> 0
arrayExample2[9] -> 0
arrayExample2[10] -> 0
arrayExample2[11] -> 0
arrayExample2[12] -> 0
arrayExample2[13] -> 0
arrayExample2[14] -> 0
```

Arrays (10)

Removing Elements from an Array

- Removing requires using a searching algorithm to find the element and remove it
- You must declare the `target` value to be removed

Searching for the Element

```
int arrayExample1[8] = {10, 9, 8, 7, 6, 5, 4, 3};
int target = 5; // Element to be removed
int i;
for(i = 0; i < (sizeof(arrayExample1) / sizeof(int)); i++) {
    if(arrayExample1[i] == target) {
      break;
    }
}
```

Removing the Element

```
if (i < (sizeof(arrayExample1) / sizeof(int))) {
    // Adjust the size of the array search
    n = (sizeof(arrayExample1) / sizeof(int)) - 1;
    // Shift our elements from the right of our found target
    for(int j = i; j < n; j++) {
      arrayExample1[j] = arrayExample1[j + 1];
    }
}
```

```
arrayExample1[0] -> 10
arrayExample1[1] -> 9
arrayExample1[2] -> 8
arrayExample1[3] -> 7
arrayExample1[4] -> 6
```

Vectors

Vectors (1)

- Similar to an array in C++ and are used to store multiple items in a single variable
- They store only a single type of data
  - i.e. double, boolean or integer
- **However**, they can grow in size dynamically
- They are considered to be:
  - **ordered**: the items have a defined order, and this order will not change when new items are added to the vector
  - **changeable**: the items of a vector are mutable (can be changed), added or removed
  - **allowable of duplicates**: vectors are indexed, and therefore items in a vector can be duplicated
- The `vector` library needs to be imported in order to use them

```
#include <vector>
```

Vectors (2)

Creating a Vector

- Vectors are created by calling the `vector` data type, followed by a set of angled brackets (`<>`)
- Inside the angle brackets the data type being stored inside the vector is specified

```
std::vector<dataType> vectorName;
```

- The items inside the vector can only be of the data type that was assigned to the variable

```
std::vector<int> intVectorExample1;    // Stores only integers
std::vector<double> dblVectorExample1; // Stores only doubles
std::vector<char> charVectorExample1;  // Stores only characters
```

Vectors (3)

Initialising a Vector i

- Vectors can be populated by using the curly braces (`{}`) after the variable name declaration

```
// Initialising List
std::vector<int> intVectorExample1 = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
std::vector<double> dblVectorExample1 = {1.0, 2.0, 3.0, 4.0};
std::vector<char> charVectorExample1 = {'a', 'b', 'c', 'd', 'e'};
```

```
// Uniform Initialising
std::vector<int> intVectorExample1 {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
std::vector<double> dblVectorExample1 {1.0, 2.0, 3.0, 4.0};
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
```

Vectors (4)

Initialising a Vector ii

- Alternatively, vectors can be created by declaring an initial size and a default value

```
std::vector<int> intVectorExample1(5, 0);
```

```
intVectorExample1[0] -> 0
intVectorExample1[1] -> 0
intVectorExample1[2] -> 0
intVectorExample1[3] -> 0
intVectorExample1[4] -> 0
```

Vectors (5)

Accessing a Vectors Element i

- The items in a vector can be accessed by referring to its index number inside a set of square brackets (`[]`)
    - *remember* that the index of a vector begins at `0` in C++

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
```

```
charVectorExample1[0] -> a
charVectorExample1[1] -> b
charVectorExample1[2] -> c
charVectorExample1[3] -> d
charVectorExample1[4] -> e
```

Vectors (6)

Accessing a Vectors Element ii

- Alternatively, elements in a vector can be accessed by using the `at()` function

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
```

```
charVectorExample1.at(0) -> a
charVectorExample1.at(1) -> b
charVectorExample1.at(2) -> c
charVectorExample1.at(3) -> d
charVectorExample1.at(4) -> e
```

- The `at()` function is preferred
  - throw an exception if trying to access an out-of-bounds index

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
```

```
charVectorExample1[5] ->
charVectorExample1.at(5) -> vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
```

Vectors (7)

Getting the Size of a Vector

- Vectors have an inbuilt function to get the size
- Achieved by calling the function `size()` on the vector

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
charVectorExample1.size();
```

```
charVectorExample1.size() -> 5
```

Vectors (8)

Inserting Elements into a Vector

- The elements in a vector are **ordered** and **indexed**, therefore modifiable
  - otherwise known as being *mutable*
- Elements can be added to a vector by using the `push_back()` function
  - this will insert an element at the end of the vector

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
charVectorExample1.push_back('f');
```

```
charVectorExample.at(0) -> a
charVectorExample.at(1) -> b
charVectorExample.at(2) -> c
charVectorExample.at(3) -> d
charVectorExample.at(4) -> e
charVectorExample.at(5) -> f
```

Vectors (9)

Replacing an Element in a Vector

- Elements at a particular index can also be replaced
- Achieve by accessing the index of the element you wish to replace using the `at()` function

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
charVectorExample1.at(2) = 'z';
```

```
charVectorExample.at(0) -> a
charVectorExample.at(1) -> b
charVectorExample.at(2) -> Z
charVectorExample.at(3) -> d
charVectorExample.at(4) -> e
```

Vectors (10)

Removing an Element from a Vector

- Elements can be removed from a vector using the `pop_back()` function
  - this will remove the *last* element in the vector

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
charVectorExample1.pop_back();
```

```
charVectorExample.at(0) -> a
charVectorExample.at(1) -> b
charVectorExample.at(2) -> c
charVectorExample.at(3) -> d
```

```
charVectorExample1.size();
```

```
charVectorExample1.size() -> 4
```

Vectors (11)

Additional Vector Functions

- These are additional functions that do not need much explanation…

Function	Description
`.clear()`	removes all elements
`.front()`	returns a reference to the first element
`.back()`	returns a reference the last element
`.empty()`	returns `1` if the vector is empty
`.capacity()`	returns the overall *capacity* of the vector

Vectors (12)

Capacity of a Vector

```
std::vector<char> charVectorExample1 {'a', 'b', 'c', 'd', 'e'};
charVectorExample1.pop_back();
```

```
charVectorExample.at(0) -> a
charVectorExample.at(1) -> b
charVectorExample.at(2) -> c
charVectorExample.at(3) -> d
```

```
charVectorExample1.size();
charVectorExample1.capacity();
```

```
charVectorExample1.size() -> 4
charVectorExample1.capacity() -> 5
```

Maps

Maps (1)

- Maps are used to store multiple items into a single variable
  - they are stored as a `key:value` pair
- A map in C++ Can be compared to a **dictionary** in Python
- They are considered to be:
  - **ordered**: the items have a defined order, and this order will not change when new items are added to the map
  - **changeable**: the items of a map are mutable (can be changed), added or removed
  - **no duplicates allowed**: maps are unable to have the same key twice
- The `map` library needs to be imported in order to use them

```
#include <map>
```

Maps (2)

Creating a Map

- Maps are created by calling the `map` data type, followed by a set of angled brackets (`<>`)
- Inside the angle brackets are the data types for the key and value

```
std::map<key_dataType, value_dataType> mapName;
```

- The items inside the map can only be of the data type that was assigned to the key and value

```
std::map<int, std::string> mapExample1;         // Stores the key as a integer, and values as a string
std::map<std::string, std::string> mapExample2; // Stores the key as a string, and values as  astring
std::map<std::string, int> mapExample3;         // Stores the key as a string, and values as integer
```

Maps (3)

Initialising a Map

- Map can be populated by using the curly braces (`{}`) after the variable name declaration

```
// Initialising List
std::map<int, std::string> mapExample1 = \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
std::map<std::string, std::string> mapExample2 = \
  {{"Lecturer 1", "Ian Cornelius"}, {"Lecturer 2", "Terry Richards"}, {"Lecturer 3", "Daniel Goldsmith"}};
```

```
// Uniform Initialising
std::map<int, std::string> mapExample1 \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
std::map<std::string, std::string> mapExample2 \
  {{"Lecturer 1", "Ian Cornelius"}, {"Lecturer 2", "Terry Richards"}, {"Lecturer 3", "Daniel Goldsmith"}};
```

Maps (4)

Accessing a Maps Elements i

- The items in a map can be accessed by referring to its *key* inside a set of square brackets (`[]`)

```
std::map<int, std::string> mapExample1 = \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
std::map<std::string, std::string> mapExample2 \
  {{"Lecturer 1", "Ian Cornelius"}, {"Lecturer 2", "Terry Richards"}, {"Lecturer 3", "Daniel Goldsmith"}};
```

```
mapExample1[0] -> Ian Cornelius
mapExample1[2] -> Daniel Goldsmith
mapExample2["Lecturer 1"] -> Ian Cornelius
mapExample2["Lecturer 3"] -> Daniel Goldsmith
```

Maps (5)

Accessing a Maps Elements ii

- Alternatively, elements in a map can be accessed by using the `at()` function

```
std::map<int, std::string> mapExample1 = \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
```

```
mapExample1.at(0) -> Ian Cornelius
mapExample1.at(1) -> Terry Richards
mapExample1.at(2) -> Daniel Goldsmith
```

- The `at()` function is preferred
  - throw an exception if trying to access an out-of-bounds index

```
std::map<int, std::string> mapExample1 = \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
```

```
mapExample1[5] ->
mapExample1.at(5) ->
```

Maps (6)

Getting the Size of a Map

- Maps have an inbuilt function to get the size
- Achieved by calling the function `size()` on the vector

```
std::map<int, std::string> mapExample1 = \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
mapExample1.size();
```

```
mapExample1.size() -> 3
```

Maps (7)

Inserting Elements into a Map i

- The elements in a map are **ordered** and **indexed**, therefore modifiable
  - otherwise known as being *mutable*
- Elements can be added to a vector by using the `insert()` function
  - used in conjunction with the `make_pair()` function

```
std::map<int, std::string> mapExample1 = \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
mapExample1.insert(std::make_pair(3, "Kabiru Mohammed")); // Alternative to mapExample1.insert({3, "Kabiru Mohammed"});
```

```
mapExample1.at(0) -> Ian Cornelius
mapExample1.at(1) -> Terry Richards
mapExample1.at(2) -> Daniel Goldsmith
mapExample1.at(3) -> Kabiru Mohammed
```

Maps (8)

Inserting Elements in a Map ii

- Alternatively, elements can be added to a vector by using the square brackets (`[]`) notation
  - a new key is provided inside the square brackets

```
std::map<int, std::string> mapExample1 = \
  {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
std::map<std::string, std::string> mapExample2 \
  {{"Lecturer 1", "Ian Cornelius"}, {"Lecturer 2", "Terry Richards"}, {"Lecturer 3", "Daniel Goldsmith"}};
mapExample1[3] = "Kabiru Mohammed";
mapExample2["Lecturer 4"] = "Kabiru Mohammed";
```

```
mapExample1.at(0) -> Ian Cornelius
mapExample2.at("Lecturer 1") -> Ian Cornelius
mapExample1.at(1) -> Terry Richards
mapExample2.at("Lecturer 2") -> Terry Richards
mapExample1.at(2) -> Daniel Goldsmith
mapExample2.at("Lecturer 3") -> Daniel Goldsmith
mapExample1.at(3) -> Kabiru Mohammed
mapExample2.at("Lecturer 4") -> Kabiru Mohammed
```

Maps (9)

Replacing an Element in a Map

- Elements at a particular index can also be replaced
- Achieve by accessing the index of the element you wish to replace using the `at()` function

```
std::map<int, std::string> mapExample1 = {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
mapExample1.at(2) = "Kabiru Mohammed";
```

```
mapExample1.at(0) -> Ian Cornelius
mapExample1.at(1) -> Terry Richards
mapExample1.at(2) -> Kabiru Mohammed
```

Maps (10)

Removing an Element from a Map

- Elements can be removed from a map using the `erase()` function
  - this will remove the element in the map with the provided *key*

```
std::map<int, std::string> mapExample1 = {{0, "Ian Cornelius"}, {1, "Terry Richards"}, {2, "Daniel Goldsmith"}};
mapExmaple1.erase(1);
```

```
mapExample1.at(0) -> Ian Cornelius
mapExample1.at(2) -> Daniel Goldsmith
```

Maps (11)

Additional Map Functions

- These are additional functions that do not need much explanation…

Function	Description
`.clear()`	removes all elements
`.find()`	searches the map for a given key and returns
`.empty()`	returns `1` if the map is empty

Goodbye

Goodbye (1)

Questions and Support

- Questions? Post them on the **Community Page** on Aula
- Additional Support? Visit the Module Support Page
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk