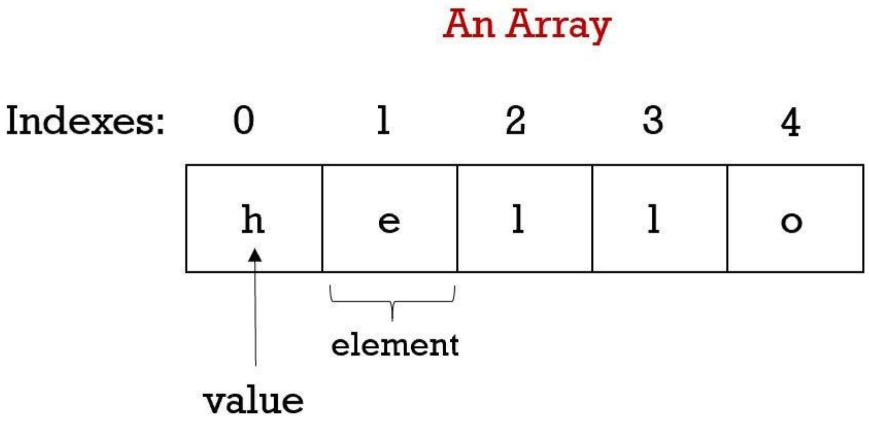
C++ Programming Bootcamp 2

COSC2802

TOPIC 2 - Arrays

TOPIC 2 | Workshop Overview

Arrays and Vectors!



Arrays and Vectors

Features

- Collection of elements of the same type
- Stored in Contiguous memory locations

- Three different versions:

1. Built in Array: fixed size, lack of support for common operations ***
2. Array Object (STL) – not in zybooks
3. Vector Object (STL) - can be resized; similar to Java ArrayList

*** We will revisit this later when we cover pointers

1. Built-in Array

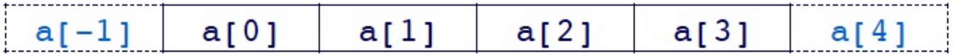
```
#define LENGTH 5
```

Can be defined in two ways:

```
int array1[LENGTH] = {0,1,2,3,4};
```

```
int array2[LENGTH];

for(int i=0; i < LENGTH; ++i) {
    array2[i] = i;
    std::cout << array2[i] << std::endl;
}
```



- Cells “before” and “after” and start/end of the array can be accessed!
- It is the programmer’s responsibility to ensure that a program does not access outside an array’s limits.

```
std::cout << array1[-1] << std::endl;
std::cout << array1[LENGTH] << std::endl;
```

These two lines will return garbage or crash your program! Be careful.

2. STL Array

Can be created using: std::array<type, size> (requires #include <array>)

STL Array provides an object-oriented version of arrays, which contain a bunch of useful functions that make using basic arrays much easier.

```cpp
std::array<int, LENGTH> std_array;
std_array.fill(0);

for(auto a : std_array) {
    std::cout << a << std::endl;
}
```

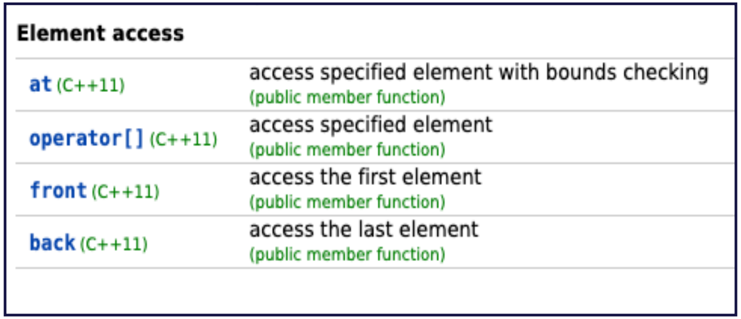
What's the difference between these two pairs of statements?

```cpp
std::cout << std_array[-1] << std::endl;
std::cout << std_array[LENGTH] << std::endl;
```

```cpp
std::cout << std_array.at(-1) << std::endl;
std::cout << std_array.at(LENGTH) << std::endl;
```

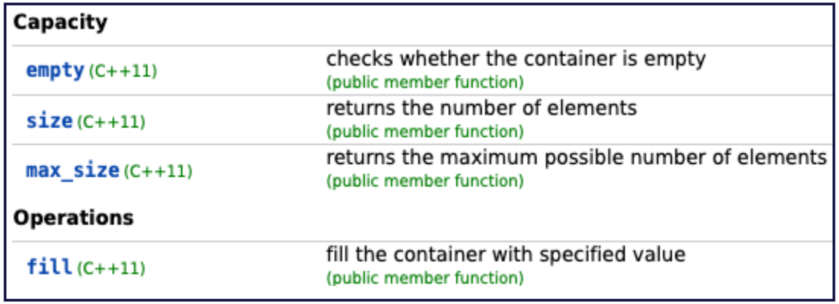
2. STL Array (contd …)

- Useful functions include - see https://en.cppreference.com/w/cpp/container/array



Element access

at (C++11)	access specified element with bounds checking (public member function)
operator[] (C++11)	access specified element (public member function)
front (C++11)	access the first element (public member function)
back (C++11)	access the last element (public member function)



Capacity

empty (C++11)	checks whether the container is empty (public member function)
size (C++11)	returns the number of elements (public member function)
max_size (C++11)	returns the maximum possible number of elements (public member function)

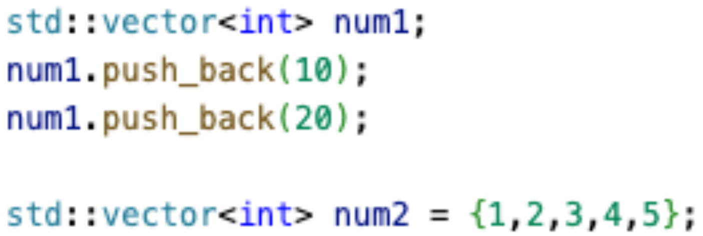
Operations

fill (C++11)	fill the container with specified value (public member function)

3. STL Vector

Vectors are similar to arrays in concept but are dynamically sized!

This means that you can add or delete or insert any number of elements as you see fit during a program.



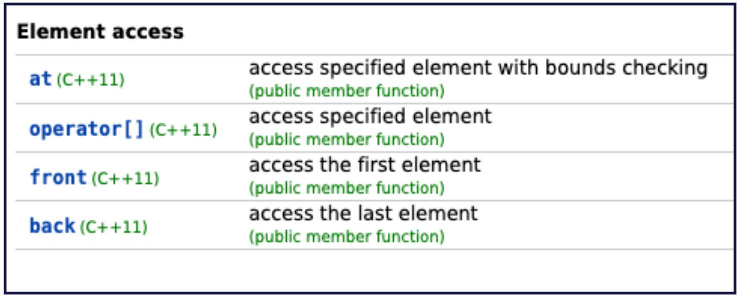
```
std::vector<int> num1;
num1.push_back(10);
num1.push_back(20);

std::vector<int> num2 = {1,2,3,4,5};
```

You can have dynamically sized arrays in C++, but it requires complex memory management, which we typically want to avoid. We will discuss this later when we get to pointers and will be used in Studio 2.

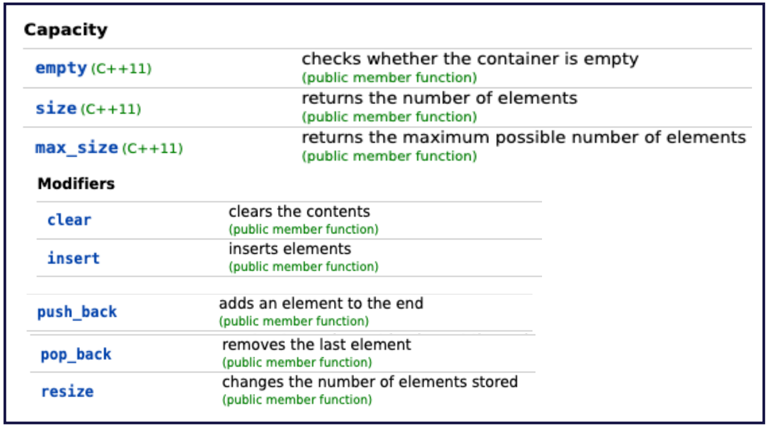
3. STL Vector (contd …)

- Useful functions include https://en.cppreference.com/w/cpp/container/vector



Element access

`at` (C++11)	access specified element with bounds checking (public member function)
`operator[]` (C++11)	access specified element (public member function)
`front` (C++11)	access the first element (public member function)
`back` (C++11)	access the last element (public member function)



Capacity

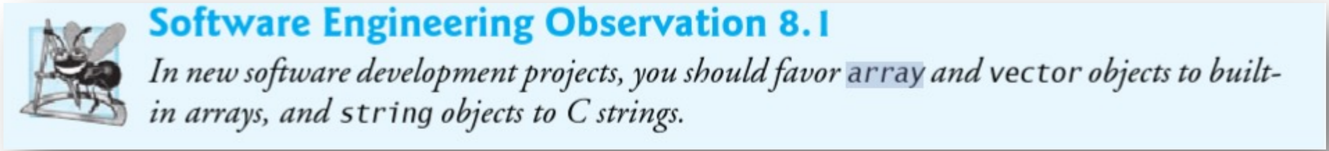
`empty` (C++11)	checks whether the container is empty (public member function)
`size` (C++11)	returns the number of elements (public member function)
`max_size` (C++11)	returns the maximum possible number of elements (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`push_back`	adds an element to the end (public member function)
`pop_back`	removes the last element (public member function)
`resize`	changes the number of elements stored (public member function)

When should you use an Array vs a Vector ??

- If you know at compile time how big your data is: use std::array
- If you do not know how much data is going to be used: use a std::vector



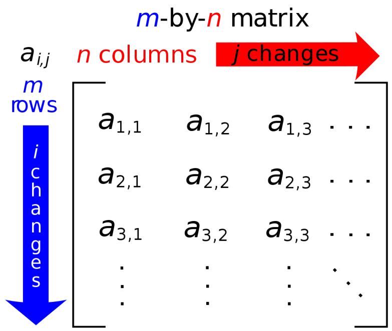
Software Engineering Observation 8.1

In new software development projects, you should favor array *and* vector *objects to built-in arrays, and* string *objects to C strings.*

- Typically, we want to avoid raw arrays as much as possible, as they just introduce unnecessary risk for development.

Multi-Dimensional Containers

Like a photograph! Which is a 2-dimensional container of pixel values!


m-by-*n* matrix
$a_{i,j}$
n columns
j changes
m rows
i changes
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $\cdots$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $\cdots$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $\cdots$
$\vdots$ $\vdots$ $\vdots$ $\ddots$

```
std::array<std::array<int, ROWS>, COLS> array_2d;

for(int c = 0; c < COLS; ++c) {
    for(int r = 0; r < ROWS; ++r) {
        array_2d[c][r] = c*ROWS + r;
        std::cout << array_2d[c][r] << std::endl;
    }
}
```

- Multi-dimensional arrays
  - Again, similar to Java

```
int a[DIM1][DIM2];
```

  - Inline initialisation is trickier

```
int a[DIM1][DIM2] = { {1,2,3}, {4,5,6}, …};
```