Programming Studio 2 – C++

Week 7 Class 2

Dr Ruwan Tennakoon
Dr Steven Korevaar

Week 7 Class 2 Summary

Types

- Auto keyword, **you must not use auto**!

Recursion

- Recursive backtracking algorithm, **you must use recursion!**

STL Containers

- Arrays, vectors, queues/deques, and lists
- Set, map, and tuples

STL Containers

C++17 Standard Template Library:

Usage in the assignment: any libraries we discuss or use in these workshops can be used in the assignment (memory, iostream, vectors, etc).

Premade classes that provide a quick and efficient access to data structures. Each provide different trade-offs in terms of computational and memory efficiency.

Sequence Containers:

Structures that hold data in sequences:

- Arrays, vectors, queues/deques, and lists

Associative Containers:

Structures that hold data in non-sequential formats.

- Set, map, and tuples

std::array

An object version of a standard C-style array.

```
int[3] a = {1,2,3};
std::array<int, 3> b = {1,2,3};
```

Provides some quality-of-life features:

- **.at():** accessing elements in the array with built in bounds checking.
- **.size():** returns the size of the array.
- **Iterators:** allows you to iterate over all elements in the array simply.

The main benefit is the very low overhead for use, like c-style arrays, and some nicer object-oriented behaviour, like storing the size of the array (with c-style arrays, this must be kept track of manually outside the object)

Cannot be resized once created!

std::vector

A much higher quality-of-life array-like data structure.

```
std::vector<int, 3> a = {1,2,3};
```

Provides a lot of quality-of-life features:

- **.at():** accessing elements in the array with built in bounds checking.
- **.size():** returns the size of the array.
- **Iterators:** allows you to iterate over all elements in the array simply.
- **Dynamically sized:** you can add or remove elements dynamically (can use .reserve() to reserve extra memory or shrink_to_fit() to reduce the memory to only what is currently required). By default, vectors reserve some additional memory, and only resizes itself when that limit is reached. So, the overhead for resizing is minimal.

std::array vs std::vector:

Arrays are statically sized, and declared on the stack by default, the programmer must allocate the array manually to the heap if they wish to dynamically resize the array. Once created the array cannot be resized as well.

Vectors automatically manage the memory of the data on the heap, automatically resizing as needed. But this costs some overhead in performance.

std::list

A pre-implemented version of a linked list.

```
std::list<int> a = {1,2,3};
```

Data is stored non-sequentially in memory.

Interaction is more limited compared to vectors and arrays: no random access via [n], operators. Getting specific elements is more difficult as the list must be traversed.

Iterable:

- `for (int n : l) std::cout << n << ", ";`

Adding new elements into lists and removing elements are trivial. For vectors and arrays, if an element is inserted in the middle of the data, all subsequent elements must be shuffled forwards to make room or and likewise backwards to fill up the empty space when an element is removed.

Resizing is also trivial, as only the memory for the new elements must be allocated when a new element is added, no pre-allocation required, as data is non-sequential.

std::deque

A mix of list and vector that provides faster insertions/deletions/resizes than vectors, while adding random access to lists.

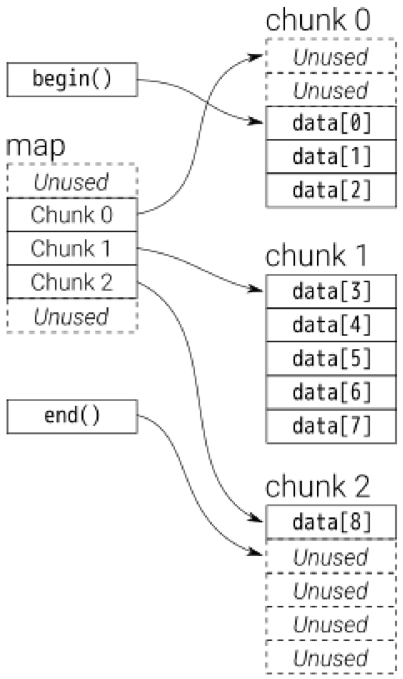
```
std::deque<int> a = {1,2,3};
```

As an interface it is almost identical to vectors, the main difference is their underlying implementation.

Vectors store all information in memory completely sequentially. Lists store all information non-sequentially in memory but have pointers from each element pointing to the next element.

Deques stores multiple "blocks" of memory, within each block the data is stored sequentially in memory, but the blocks themselves are not sequential.

This approach allows for both random access, and faster insertion and deletion. But at a small cost to both. Random access is not as fast as vectors, and insertion/deletion is not as fast as lists.



RMIT UNIVERSITY

std::map

A map can be thought of as an array that is indexed by a “key” instead of sequential integer indices.

```
// Create a map of strings to integers
std::map<std::string, int> map;

// Insert some values into the map
map["one"] = 1;
map["two"] = 2;
map["three"] = 3;
```

Values can be accessed later by using the [] operator and the key as the index.

```
std::cout << map["one"] << std::endl;
// Prints: 1
```

RMIT UNIVERSITY

std::set

A set is a collection of unique objects.

```
std::set<int> setOfIntegers = {1,2,3,4,4,5,5};
//creates {1,2,3,4,5}
```

Useful for keeping a track of unique elements, where duplicates must be avoided.

Usually iterated through for each syntax.

```
for(int i : setOfIntegers) {
    std::cout << *it << " ";
}
std::cout << std::endl;
```

Main use case is determining if elements are present in the set or not. Which can be used by calling .count(<element>) with an instance of the element as an argument.

```
std::set<int> setOfIntegers = {1,2,3,4,5};
std::cout << setOfIntegers.count(3) << std::endl;
```

RMIT UNIVERSITY

std::tuple

A tuple is a temporary data structure that can hold a sequence of data that has differing types.

```
// Making a tuple of characters, integers, and floats
tuple <char, int, float> tup;
tup = make_tuple('a', 1, 1.1);
```

You can both get and set values using the syntax: “`get<index>(variable_name)`”

```
get<0>(tup) = ‘b’
std::cout << get<0>(tup) << std::endl;
// Prints: ‘b’
```

Recursion

Solving a problem, by using the solution for a smaller subset of the problem with a small additional step.

E.G., a solution to a problem with input n, can be solved by finding the solution for the input $n - 1$ plus some additional small value. i.e.:

$$P(n) = P(n-1) + X$$

Typically, we need to define a base case: such as $\mathrm{P}(n) = 0$, such that all future cases can built off that base case.

```
int factorial(int num){
    int retValue = 1; //base case, 0! = 1
    if(num != 0) {
        //recursive or inductive step
        retValue = num * factorial(num-1);
    }

    return retValue;
}
```

Flood Fill

What is flood fill?

Filling a space, imagine dropping water, the water spreads out over time eventually filling up a container.

How does it work?

We can take a recursive approach:

Each time the "flood fill" function is called on a cell, it will "fill" the surrounding cells and call the flood fill function on its neighbour cells.

Download the starter code from canvas.

Week 7 Class 2 Exercises

Think about the types of containers we covered today,

- for what kind of problems are they good for?
- Efficiency: computation vs memory

Check the Canvas modules -> week 7 -> class 2 -> class notes pdf for more exercises regarding recursion.

Continue working on the assignment.

Prepare for checkpoint 3: covers milestone 2

- Reading the maze from command line,
- Building the maze in Minecraft,
- cleaning up after,
- solving the maze.