Programming Studio 2
COSC2804

Week 07 - Class 02 Notes

Ruwan Tennakoon
ruwan.tennakoon@rmit.edu.au

Steven Korevaar
steven.korevaar@rmit.edu.au

Introduction

We will explore the following key concepts in the class:

- Recursion – revision
- STL
  - Sequential Containers
  - Associative Containers
  - Container Iterations

Tutorial

1. We will implement a flood fill algorithm in class as the example: Fill an area that is at the same height as a given location using Lime carpet.
   - To get started, download the starter code and study it (`main.cpp`).
   - Start planning the implementation - recursion. Using the static reasoning method we discussed last class, work out what will be the base case and the recursive step.
     There are couple of ways of doing this, We will use one such rational - You may do it differently.

```
// Base-case 1: not same elevation
    // Do nothing
// Base-case 2: already filled
    // Do nothing
// Recursive step:
    //          Update the block at the current location
    //          FloodFill for adjacent cells [up, down, left, right]
```

   - Because our base cases involve comparing with the previous step, Let us write a new function. This will be our recursive function:

```
void Field::floodFill(mcpp::Coordinate curr, mcpp::Coordinate prev,
                                                mcpp::BlockType block,
                                                mcpp::MinecraftConnection& mc)

{
// Base-case 1: not same elevation
    // Do nothing
// Base-case 2: already filled
    // Do nothing
// Recursive step:
    //          Update the block at the current location
    //          FloodFill for adjacent cells [up, down, left, right]
}
```

   - Let us now implement the function

```
if(mc.getHeight(curr.x, curr.z) != mc.getHeight(prev.x, prev.z)){
    //Base-case 1: not same elevation
    //      do nothing
}
```

```
else if(mc.getBlock(curr) == block){
    //Base-case 2: already filled
    //      do nothing
}
```

```cpp
else{
    //Recursive step
    mc.setBlock(curr, block);

    mcpp::Coordinate xPlus(curr.x+1, mc.getHeight(curr.x+1, curr.z)+1 ,curr.z);
    floodFill(xPlus, curr, block, mc);

    mcpp::Coordinate xMinus(curr.x-1, mc.getHeight(curr.x-1, curr.z)+1 ,curr.z);
    floodFill(xMinus, curr, block, mc);

    mcpp::Coordinate zPlus(curr.x, mc.getHeight(curr.x, curr.z+1)+1 ,curr.z+1);
    floodFill(zPlus, curr, block, mc);

    mcpp::Coordinate zMinus(curr.x, mc.getHeight(curr.x, curr.z-1)+1 ,curr.z-1);
    floodFill(zMinus, curr, block, mc);

}
```

- Now complete the FloodFill function in the interface.

```cpp
void Field::floodFill(mcpp::Coordinate start, mcpp::BlockType block){
    mcpp::MinecraftConnection mc;
    floodFill(start, start, block, mc);
}
```

- What are the assumptions - work out the contract:

```cpp
/*
Floodfill
contract: the start location should not contain the
same block type as block
*/
void floodFill(mcpp::Coordinate start, mcpp::BlockType block);
```

Exercise: See if you can come up with a better way to do the recursive function - allows you to get rid of the assumption.

2. Standard Template Libraries (STL): A set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.

   The C++ STL provides a number of containers to represent common and useful data structures

   - **Sequence Containers**: Store elements of the container is a defined sequence. The order of the containers is important. The implementations guarantee the order of the container.
     - Array
     - Vector
     - Deque
     - List
   - **Associative Containers**: Provide relationships (associations) between elements of the container. Containers are un-ordered.
     - Set
     - Map
     - Tuple

   (a) Start with `array` (`containers.cpp`):

```
#include <array>
std::array<int, 3> array({1,2,3});

std::cout << "array[0]: " << array[0]
        << ", array[1]: " << array[1]
        << std::endl;

array[0] = 100;

std::cout << "array[0]: " << array[0]
        << ", array[1]: " << array[1]
        << std::endl;
```

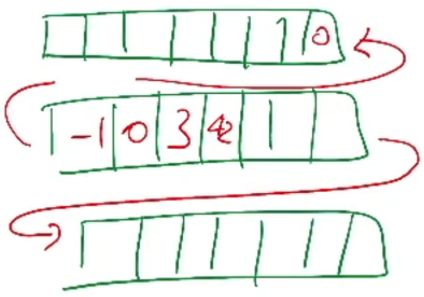
Exercise: See how `std::vector` and `std::list` work

(b) **Deque**: Combines benefit of all data structures above - Expandability – constant time – like LinkedList (both add back and add front). Access front and back – constant time – like all above. Random access – constant time – like array and vector.

```
#include <deque>
std::deque<int> deque;
deque.push_back(1);
deque.push_back(2);
deque.push_back(3);

deque[0] = 100;
std::cout << "deque[0]: " << deque[0]
        << ", deque[1]: " << deque[1]
        << std::endl;
```

Deques are implemented as 2D arrays. Every time you run out of space – add another array (1d) and link in the second dimension.



(c) **Set**: As in the mathematical sense. Each item in the set is unique. Requires the type of the set to have a well-defined operation for determining if two elements of the set are unique

```
#include <set>
std::set<int> set;
set.insert(1);
set.insert(2);
set.insert(3);

std::cout << "size: " << set.size() << std::endl;
set.insert(2);
std::cout << "size: " << set.size() << std::endl;

//find if an element in the set
// std::cout << "inSet(1): " << set.find(1) << std::endl; // does not work
```

```
bool inSet = set.find(1) != set.end();
std::cout << "inSet(1): " << inSet << std::endl;

inSet = set.find(100) != set.end();
std::cout << "inSet(100): " << inSet << std::endl;
std::cout << "inSet(2): " << *(set.find(2)) << std::endl;
```

(d) **Iterators**: An iterator is a generalised pointer into a container

```
std::cout << "Printing Vector" << std::endl;
for(int i =0; i != vector.size(); ++i){
    std::cout << "\t vector[" << i << "]: " << vector[i]
        << std::endl;
}
std::cout << "Done Printing Vector" << std::endl;
```

Difficult to do for some other containers. There is a better way through c++ STL

```
for(int& value : vector){
    std::cout << "\t vector[" << "?" << "]: " << value
        << std::endl;
}
```

Reference allows to change - this is not okay for set (associative containers) - need `const` keyword.

```
for(const int& value : set){
    std::cout << "\t set[" << "?" << "]: " << value
        << std::endl;
}
```

3. **Type System**: For a high-level language, the type system underpins the formal mathematics of how the language: Is compiled, Is evaluated.

   - In a **Strongly typed** language, the type of an entity cannot be changed, once the types of the entity has been instantiated. Languages include: C++, Java, Python.
   - In a **Weakly typed** language, the type of an entity can be changed. Languages include: Perl, Javascript.

   Static vs Dynamic Typing:

   - In a **Statically typed** language, the type of an entity is determined at compilation. Languages include: C++, Java (Except for some forms of typecasting).
   - In a **Dynamically typed** language, the type of an entity is determined at runtime. Languages include: Python, Perl, Javascript

   Need to be careful. Change `auto.cpp` all to auto and run:

```
float b = 6.8;

//will do implicit typecast to double not float
auto b = 6.8;

//can force typecasting by
auto b = 6.8f;

//Issue: compiler does not look for the entire code to determine the
// appropriate type. Just look at the initialization.
```

```cpp
auto x = 1;
std::cout << x/2 << std::endl;
```

Exercises

1. Create `std::vector` and `std::list` and store some data as done above to see how they work:

```cpp
#include <vector>

std::vector<int> vector;
vector.push_back(1);
vector.push_back(2);
vector.push_back(3);

vector[0] = 100;
std::cout << "vector[0]: " << vector[0]
        << ", vector[1]: " << vector[1]
        << std::endl;
```

```cpp
#include <list>

std::list<int> list;
list.push_back(1);
list.push_front(2);
list.push_back(3);

//list[0] = 100; //No random access
std::cout << "list.front: " << list.front()
        << ", list.back: " << list.back()
        << std::endl;
```

2. Do the same with **Map**: Associates pairs of elements. Associates by key-value. The key is the primary entity. Each key has an associated value.

```cpp
#include <map>
std::map<int, std::string> map;
map[5] = "hello";
map[-1] = "world";
map[0] = "!";
map[5] = "cosc2804";
std::cout << "map[-1]" << map[-1] << std::endl;
std::cout << "map[5]" << map[5] << std::endl;

bool inMap = map.find(5) != map.end();
std::cout << "inMap(5): " << inMap << std::endl;

//Can we dereference?
//std::cout << "inMap(5): " << *(map.find(5)) << std::endl;
//std::pair<int, std::string>& resultFind = *(map.find(5)) ;
std::pair<const int, std::string>& resultFind = *(map.find(5)) ;
std::cout << "Map(5): " << resultFind.second << std::endl;
std::cout << "Map(5): " << map.find(5)->second << std::endl;
```

3. Now try Tuple: Associates a group of elements together Association has no defined “key”. Only associates 1 set of elements. In general, permits an n-tuple, where n is the user choice.

```
#include <utility>
std::tuple<double, char> tuple;
std::get<0>(tuple) = 7.1f;
std::get<1>(tuple) = 'a';

std::get<0>(tuple) = 10.2f;
std::cout << "(" << std::get<0>(tuple) << ", " << std::get<1>(tuple)
            << ")" << std::endl;
```

4. Change the flood fill implementation to be more readable by using containers and iteration.

```
// initialise in constructor
std::vector< mcpp::Coordinate > neighbourhood;
neighbourhood.push_back(mcpp::Coordinate(1,0,0));    //up
neighbourhood.push_back(mcpp::Coordinate(-1,0,0));   //down
neighbourhood.push_back(mcpp::Coordinate(0,0,1));    //right
neighbourhood.push_back(mcpp::Coordinate(0,0,-1));   //left

//Recursive step
mc.setBlock(curr, block);
for(mcpp::Coordinate& offset : neighbourhood){
    mcpp::Coordinate next(
                            curr.x+offset.x,
                            mc.getHeight(curr.x+offset.x, curr.z+offset.z)+1 ,
                            curr.z+offset.z);

    floodFill(next, curr, block, mc);
}
```

5. Implement a function to flatten a rectangular region in Minecraft world using recursion.