Tute 9: Containers & Standard Library

How can the sidewinder algorithm be used to generate a maze?
Please see the algorithm outlined here: http://weblog.jamisbuck.org/2011/2/3/maze-generation-sidewinder-algorithm

Review: STL

What are the main components of the Standard Library?

As well as the standard C headers, the standard library can be divided into the following sections:

- Containers: various containers used to store data.
- General: includes algorithm, chrono, memory, iterator, utility, string, regex.
- Threading: atomic, mutex, thread
- Streams: various stream headers.

The standard library has different implementations among different compilers, but places requirements on specific syntax used, as well as requirements on performance (performance requirements often correspond to a well-known algorithm or data structure, which is expected but not required to be used).

What containers exist in the STL?

A container is a data structure which is implemented irrespective of the data that will be stored inside it (ie, a class template can be used to create a container). The main types of containers are:

- Sequence containers: these are containers that maintain the original ordering of inserted elements:
  - vector
  - deque
  - forward_list
  - list
  - array
- Associative containers: these are containers where the elements are inserted in a pre-defined order, determined by the container's implementation (typically a type of tree, usually red-black):
  - set
  - multiset
  - map
  - multimap

- Unordered associative containers: similar to associative containers, except they are usually implemented using a hash table instead of a tree:
    - unordered_set
    - unordered_multiset
    - unordered_map
    - unordered_multimap
- Container adapters: these are a variation of one of the above containers, usually a wrapper container built on top of an existing one:
    - stack
    - queue
    - priority_queue

What is an initializer list?

the compiler will automatically construct a non-empty object of this class template type whenever an initializer list expression needs to be passed or copied, ie:

```
{1, 2, 3, 4, 5, 6, 7};
{“hello”, “world”};
```

The types of these objects are initializer_list. These can also be assigned, and if the auto keyword is used, the compiler will assume the type is initializer_list:

```
auto a = {1, 2, 3, 4, 5, 6, 7};
```

What is uniform initialization?

Uniform initialization is when an initialization list is used to populate a container with elements, eg:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7};
```

This is the same as object construction you have learned about previously, in this case, the vector has a constructor that takes an argument of type initializer_list, and that is what is being used to construct the vector object here.

What is an iterator and where are they used?

An std iterator is typically any object that points to an element in a container and has the ability to iterate through the elements of that container using the ++ operator, and access an element using the * operator. Not all iterators in the standard library have the same functionality, the different types are (in increasing order of functionality, ie random-access has all the functionality of bidirectional):

- input/output
- forward
- bidirectional
- random-access

Programming Concepts: Iterators

How can we make use of custom iterators?

In order for a container to use an iterator, it must have a begin() and end() method that returns an iterator, which in turn is an object that must overload the ++, * and != operators. These can then be used to manually iterate through a container:

```
for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++)
	std::cout << *it;
```

or implicitly:

```
for (auto i : v)
	std::cout << i;
```

What is a comparison functor and how can it be used with an std::set to aid pathfinding?

An std::set can be used to store the closed set for the breadth and a* searches (and for an iterative depth first). By default this will perform a < comparison on whatever is stored in the set to determine the internal ordering. If you need to perform a different comparison to the default one, you must define your own function object (functor):

```
template <typename T>
class NodeCmp : std::less<T>
{
public:
	bool operator () (const T *left, const T *right) { return left->node < right->node; }
};
```

This is an object that will run the () operator when executed as a function. It is also templated (which we will learn more about next week).

The std::set can be initialised using this functor like so:

```
        std::set<T*, NodeCmp> closedSet;
```

Compilation: Dependencies

What is a dependency file?

A dependency is used to determine when a target needs to be rebuilt. if the target's modification time is older than that of its dependencies, the target is rebuilt according to a defined build rule. A dependency file is a file that saves this information, so that it can be looked up automatically at compile time (refer to assignment 1 sample solution for an example).

How can dependency files be generated and used?

A list of dependency files can be generated in the same way as a list of object files. The compile command can then be modified by adding the following flags:

- -MP: Adds phony includes for headers that depend on nothing (required in case headers are removed)
- -MMD: create the actual dependency files

The dependency files can then be included in the Makefile:

```
        -include $(DEPS)
```

Assuming DEPS is the variable storing the names of the dependency files.

Errors: None this week

Debugging: Breakpoints

What is a breakpoint?

A breakpoint is used in a debugger (such as gdb) to set a location where execution should pause, so that variables can be analysed etc before continuing.

How can breakpoints be used with gdb?

When running a program in gdb, before typing the run command, you can set a breakpoint using the following:

```
break filename:line_no
```

This will cause the program to pause at that location during execution. The code can then be analysed in the regular way by using commands such as bt, f, p, etc. You can then continue execution using the c command, or step through the program line by line using the command s.

Exercises:

Create a custom array container that stores strings, as well as an iterator that can be used to implicitly iterate through the elements of the container.

```
#include <string>
#include <iostream>

class Array;

class ArrayIterator
{
private:
    int pos;
    Array *array;

public:
    ArrayIterator(Array *array, int pos = 0) : array(array), pos(pos) {}

    const ArrayIterator &operator ++ () { pos++; return *this; }
    bool operator != (const ArrayIterator &iter) const { return iter.pos != pos; }
    std::string operator * ();
};

class Array
{
        friend class ArrayIterator;
private:
    std::string data[100];
    int last = 0;

public:
    Array(std::initializer_list<std::string> list)
    {
        for (auto &s : list)
            data[last++] = s;
    }

    ArrayIterator begin() { return ArrayIterator(this); }
    ArrayIterator end() { return ArrayIterator(this, last); }
};

std::string ArrayIterator::operator * ()
{
    return array->data[pos];
}

int main(int argc, char **argv)
{
    Array array = {"Hello", "World"};
    for (auto s : array)
        std::cout << s << "\n";
    return 0;
}
```