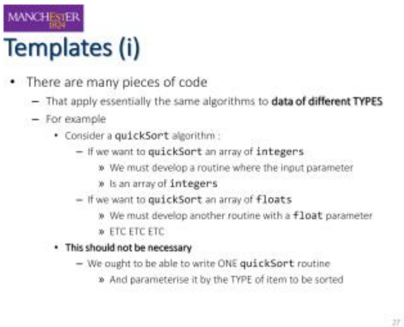
Briefing Session 2

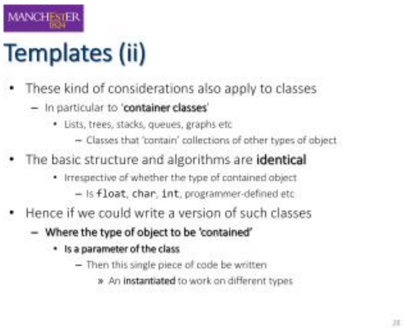
09 November 2021 09:56

The presentation starter with a discussion of C++ templates based on the slides below.
Templates are NOT part of the unit and so are not examinable. However, in using maps, mutexes etc, templates are important, so the material here is intended to give you an idea of what templa2tes are for, how they expressed and how they are instantiated. YOU WILL NOT BE ASKED TO WRITE TEMPLATES IN THIS UNIT. However, you will need to instantiate temeples from the C++ STL (Standard Template Library).



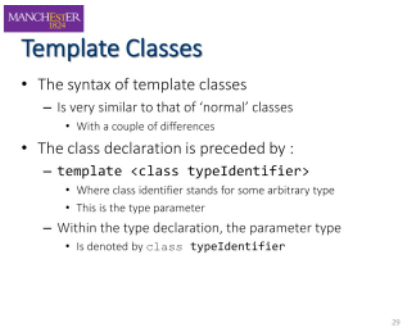
Templates (i)

- There are many pieces of code
  - That apply essentially the same algorithms to **data of different TYPES**
  - For example
    - Consider a `quickSort` algorithm :
      - If we want to `quickSort` an array of `integers`
        - » We must develop a routine where the input parameter
        - » Is an array of `integers`
      - If we want to `quickSort` an array of `floats`
        - » We must develop another routine with a `float` parameter
        - » ETC ETC ETC
    - **This should not be necessary**
      - We ought to be able to write ONE `quickSort` routine
        - » And parameterise it by the TYPE of item to be sorted



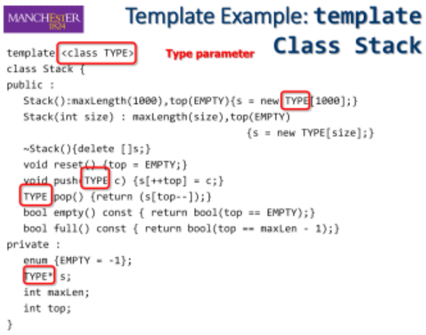
Templates (ii)

- These kind of considerations also apply to classes
  - In particular to '**container classes**'
    - Lists, trees, stacks, queues, graphs etc
      - Classes that 'contain' collections of other types of object
- The basic structure and algorithms are **identical**
    - Irrespective of whether the type of contained object
      - Is `float`, `char`, `int`, programmer-defined etc
- Hence if we could write a version of such classes
  - **Where the type of object to be 'contained'**
    - **Is a parameter of the class**
      - Then this single piece of code be written
        - » An **instantiated** to work on different types



Template Classes

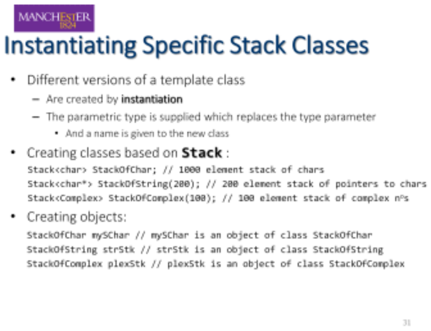
- The syntax of template classes
  - Is very similar to that of 'normal' classes
    - With a couple of differences
- The class declaration is preceded by :
  - `template <class typeIdentifier>`
    - Where class identifier stands for some arbitrary type
    - This is the type parameter
  - Within the type declaration, the parameter type
    - Is denoted by `class typeIdentifier`



Template Example: `template Class Stack`

Type parameter

```
template <class TYPE>
class Stack {
public :
   Stack():maxLength(1000),top(EMPTY){s = new TYPE[1000];}
   Stack(int size) : maxLength(size),top(EMPTY)
                                          {s = new TYPE[size];}
   ~Stack(){delete []s;}
   void reset() {top = EMPTY;}
   void push(TYPE c) {s[++top] = c;}
   TYPE pop() {return (s[top--]);}
   bool empty() const { return bool(top == EMPTY);}
   bool full() const { return bool(top == maxLen - 1);}
private :
   enum {EMPTY = -1};
   TYPE* s;
   int maxLen;
   int top;
}
```



Instantiating Specific Stack Classes

- Different versions of a template class
  - Are created by **instantiation**
  - The parametric type is supplied which replaces the type parameter
    - And a name is given to the new class
- Creating classes based on **`Stack`** :

```
Stack<char> StackOfChar; // 1000 element stack of chars
Stack<char*> StackOfString(200); // 200 element stack of pointers to chars
Stack<Complex> StackOfComplex(100); // 100 element stack of complex nºs
```

- Creating objects:

```
StackOfChar mySChar // mySChar is an object of class StackOfChar
StackOfString strStk // strStk is an object of class StackOfString
StackOfComplex plexStk // plexStk is an object of class StackOfComplex
```

This part of the session was about maps. A map is a pair - a key and a mapped value. The key is provided in order to access the mapper value. In this assignment key is thread id which can easily be found via std.this_thread, and retruns the Competitor object associated with the thread.

This is a simple example of mapping between Roman numerals (the key) and the text name of the decimal equibvalent

Roman numeral (key)	Text decimal number (mapped value)
i	one
ii	two
iii	three
iv	four
v	five
vi	six
vii	seven
viii	eight
ix	nine
x	ten

```
#include <iostream>
#include <string>
#include //other .h files

const int NO_TEAMS = 4;     // number of teams in the race
const int NO_MEMBERS = 4;    // number of athletes in the team

void run(Competitor& c) {
   // store thread id and competitor in a map
   // delay for random period
   // print message stating which competitor has just finished
}

int main() {
    thread theThreads[NO_TEAMS][NO_MEMBERS];
    Competitor teamsAndMembers[NO_TEAMS][NO_MEMBERS];
    // define elements of teamsAndMembers
    // create threads (elements of theThreads)
    // join threads
}
```

Thread id	Competitor
x	Jamacia, Bolt
a	Italy, Patta
f	Italy, Tortu
w	UK, Hughes
…	…

Like most classes in the STL, maps have many member functions. However, for this assignment you will only need to use[1]:

- `begin()` – Returns an **iterator** to the first element in the map
- `end()` – Returns an **iterator** to the notional element that follows last element in the map
- `size()` – Returns the number of elements in the map
- `insert(keyvalue, mapvalue)` – Adds a new pair to the map
- `find(keyvalue)` – Returns an **iterator** that indicates the map entry containing the key value. If the key value is not present in the map, find returns an iterator to end() .

An iterator can be thought of as a **pointer** which can be moved to point to each map element in turn. Hence iterators can be used to search for an entry (as with the `find` function).

[1] See https://thispointer.com/stdmap-tutorial-part-1-usage-detail-with-examples/

```
#include <map>
#include "Competitor.h"
...
class ThreadMap {
private:
    std::map <std::thread::id, Competitor> threadComp;
    public:
       ThreadMap();
       void insertThreadPair(Competitor c);
       Competitor getCompetitor();
       void printMapContents();
       int ThreadMapSize();
};
```

The following is **part** of `ThreadMap.cpp`:

```
#include "ThreadMap.h"

ThreadMap::ThreadMap() {}; // constructor

void ThreadMap::insertThreadPair(Competitor c) {
     // create a threadID, Competitor pair using a call to std::make_pair
     // store the pair in the map using the map insert member function
  }

Competitor ThreadMap::getCompetitor() {
   std::map <std::thread::id, Competitor>::iterator it = threadComp.find(std::this_thread::get_id());
   if (it == threadComp.end())
       return Competitor::makeNull();
   else
      return it->second;    // the second item in the pair (the Competitor)
}

void ThreadMap::printMapContents() {
     std::cout << "MAP CONTENTS:" << std::endl;
```

```
    std::map <std::thread::id, Competitor>::iterator it = threadComp.begin();
    // you need to write the rest!
    cout << "END MAP CONTENTS" << endl;
}

int ThreadMap::ThreadMapSize() { return threadComp.size(); }
```

Thread Safety

Competitor?
ThreadMap?

Are these classes thread safe i.e. can they be undated by multiple threads without interference?

Sadly the answer is NO. It is one of your jobs to work out why and to decide what to do about it.