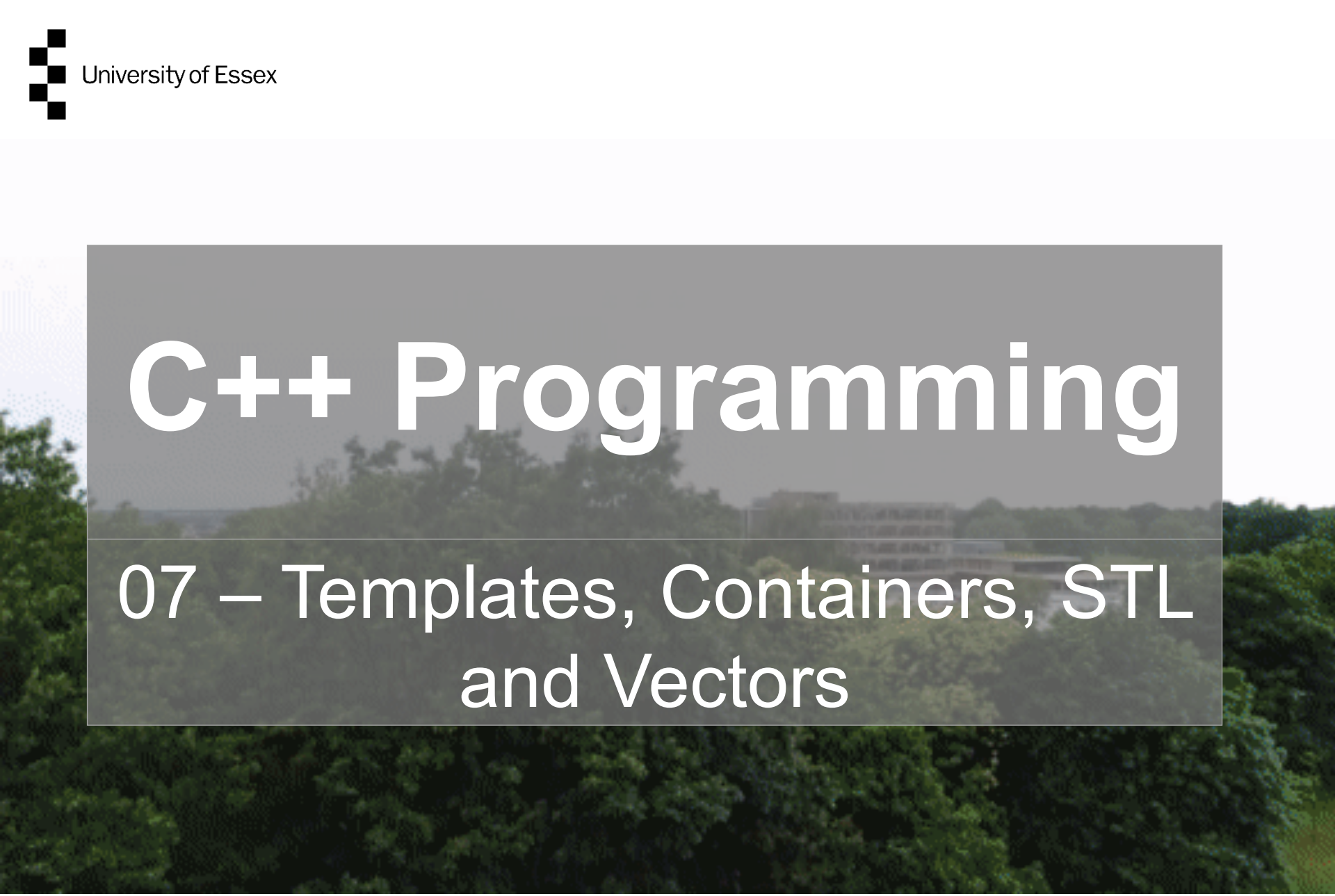
University of Essex



C++ Programming

07 – Templates, Containers, STL and Vectors

Template Functions

- Consider the following function.

```
1  void swap(int &a, int &b)
2  {
3      int temp = a;
4      a = b;
5      b = temp;
6  }
```

- If we wished to swap the values of two variables of type **float** or of type **string** we would have to write extra versions of the function with different argument types.
- It is possible to avoid the need to write overloaded versions of functions by using a ***template*** function.

Template Functions

- Here is a template version of the function from the previous slide:

```
1  template <class T>
2  void swap(T &a, T &b)
3  {
4      T temp = a;
5      a = b;
6      b = temp;
7  }
```

- The template declaration says that **T** is a type parameter that can be instantiated with any type (it doesn't have to be a class).
- If we make a call **swap(x, y)** where **x** and **y** are variables of type **float**, the compiler will generate a version of the function with **T** instantiated to **float**.

Template Functions

- The compiler will generate a separate instance of the template function for each type for which a call is made, so if we made calls **swap(x, y)** and **swap(a, b)** where **x** and **y** are of **float** and **a** and **b** are of **int** two separate versions of the function will be created.
- The machine code would hence be the same as if the programmer had written overloaded versions of the function.
- The function on the previous slide cannot be called with two arguments of different types even if a conversion between the two types exists.

Template Functions

- We could write a template function to print the contents of an array in a desired format:

```
1  template <class T>
2  void printArray(const T array[], int count)
3  {
4      for (int i = 0; i<count; i++)
5          cout << array[i] << " ";
6  }
```

- If a call is made to this function with the first argument being an array of objects of some class for which the **<<** operator has not been overloaded the compiler will report an error.

Template Functions

- A template function can have more than one type parameter, e.g.

```
1  template <class S, class T>
2  void printPair(const S &s, const T &t)
3  {
4      cout << '<' << s << ', ' << t << '>';
5  }
```

- Here **S** and **T** may be instantiated with the same type or with different types.

Template Classes

- A ***template class*** is a class in which one or more of the members has a parameterised type, e.g.
- The following class can store pairs of objects.

```
1  template <class S, class T>
2  class Pair
3  {
4      private:
5          S s;
6          T t;
7      public:
8          Pair(S a, T b): s(a), t(b) { };
9          S getFirst() const { return s; };
10         T getSecond() const { return t; };
11 };
```

Template Classes

- A variable to hold an object of this class would be declared using the syntax

Pair<string, int> a("fred", 35);

- If the complete definitions of **getFirst** and **getSecond** were in a file called **Pair.cpp** the compiler when compiling this file would not know what instantiations of **S** and **T** are required by code in other files that uses the class and hence could not generate appropriate versions of the functions.

The Standard Template Library

- The ***standard template library*** (STL) provides a number of template classes for various kinds of collections of objects, and algorithms that can be applied to these classes.
- Each of these template classes is known as a ***container***.
- There are three ***sequence containers*** to store sequential data: **vector**, **deque** and **list**.
- In addition there are three ***container adaptors***: **stack**, **queue** and **priority_queue**;

The Standard Template Library

- There are four ***associative containers*** to store non-sequential data with rapid searching: **set**, **multiset**, **map** and **multimap**.
- The STL also contains two other non-template classes, **bitset** and **valarray**, that have some of the functionality of containers, but we will not use these in this module.

The Standard Template Library

- All container classes and container adaptors have member functions called **size** and **empty**.
- There is also a function **max_size** which returns the maximum possible size for the container.
- Each class has a **swap** function, which takes as an argument a reference to another container of the same type: after a call to **c1.swap(c2)**, **c1** will contain the previous contents of **c2** and **c2** will contain the previous contents of **c1**.

The Standard Template Library

- Each class other than **priority_queue** also has the usual six comparison operators, although the behaviour of operators such as **<** depends on the container.
- The **clear** function (which takes no argument and returns no result) resets the container to be empty.
- There are also functions **begin**, **end**, **rbegin** and **rend** which can be used to obtain ***iterators*** to traverse through the elements of a collection and **erase** to remove one or more elements (using an iterator).

The vector Class

- The **vector** template class provides an implementation of arrays with range-checking and should be used when we want to access elements by position.
- Programs that use this class should contain the line **#include <vector>**.
- The no-argument constructor will initialise a vector to have no contents and a default capacity.
- There is also a constructor with an argument of type **int** which initialises the vector to contain a fixed number of elements, each initialised to the same value – this value may be specified in a second argument which defaults to 0 (or the equivalent for any other type):

```
vector<string> v1;
vector<int> v2(20); // will contain 20 zeroes
vector<char> v3(10, 'x'); // will contain 10 'x's
```

The vector Class

- For a vector of class objects the use of the single-argument **vector** constructor will cause initialisation of the objects in the vector to be performed by the class's no-argument constructor.
- In addition some other **vector** methods make use of the no-argument constructor so when writing a class that is expected to be used in collections a no-argument constructor should normally be provided.

The vector Class

- It is possible to initialise the contents of a vector to be a copy of the contents of an array.
- This is done using a constructor with two arguments – the first is the address of the beginning of the array and the second is the address of the memory location immediately after the end of the array.
- We can use a smaller second argument or a larger first argument to copy part of the array into the vector.

```
int a[] = { 1, 2, 3, 4, 5, 6, 7 };
vector<int> v4(a, a+7);
vector<int> v5(a, a+3); // will contain 1,2,3
vector<int> v6(a+4, a+7); // will contain 5,6,7
```

The vector Class

- As with the **string** class, elements of a vector can be accessed using a member function **at** or via subscripting; only the **at** function provides range-checking so for safe access to the elements of a vector an expression such as **v.at(n)** should be used.
- There are member functions **front** and **back** which return references to the first and last element; these can be used as the left-hand operand of an assignment statement:

v2.front() = 7; cout << v2.back();

- If the vector is empty a call to either of these two functions will throw an exception.

The vector Class

- There are member functions **push_back** which will append an element to the end of the vector and **pop_back** which will remove the last element (or throw an exception if the vector is empty).
- If a call to **push_back** results in the size of the vector becoming greater than the current capacity the capacity will be doubled.
- Hence if we know that many calls to **push_back** will be made it can sometimes be more efficient to increase the capacity once using **reserve** before making any of the calls:

v.reserve(100); // increase capacity to 100

The vector Class

- The **reserve** function can also be useful if the size of a large vector is about to reach its capacity and we know that we are going to add only a small number of new elements.
- We can prevent the automatic doubling of the capacity (which would result in a large amount of unused space being allocated) by reserving enough space for the extra elements, e.g.

v.reserve(v.size()+10);

The vector Class

- It is possible to change the size of a vector using **resize**.
- If the new size is smaller than the original size the vector will be truncated; if it is greater multiple copies of an extra value will be appended to the end of the vector.
- The second argument, if provided, specifies this value; the default is 0 or its equivalent, (so we must provide the second argument for a vector of objects of a class without a no-argument constructor):

```
v2.resize(5);
// elements after v2[4] will be deleted

v1.resize(10);
// empty strings appended to vector

// if its size is less than 10

v1.resize(15, "hello")
// 5 copies of "hello" appended to vector
```

The vector Class

- It is possible to insert values into locations other than the end and remove values from arbitrary locations.
- These operations involve shifting of the existing elements to the right of the insertion or deletion point and are hence rather inefficient.
- Hence if they are to be performed frequently it would be more appropriate to use a different container class such as **list** or **deque**.

Range-based For Loops

- Range-based for loops similar to the enhanced for loops of Java were introduced in C++11. The syntax is

  for (*T* i: *range*) *statement*

- The range may be an array or a container (including a string), or any other class that has been written that supports iterators.
- The statement may of course be a sequence inside **{** and **}** .
- We could output the contents of a vector one item per line using

  for (int i: v2) cout << i << endl;

Range-based For Loops

- To convert all the characters in the vector **v1** to upper-case we would use

```
for (char &c: v1) c = toupper(c);
```

- To output details of all students in a vector of students (assuming a **<<** operator has been written for the **Student** class) we could use a loop such as

```
for (Student s: studs) cout << s << endl;
```

- However this will result in copies of the student objects being made, so it would be more efficient to use

```
for (Student &s: studs) cout << s << endl;
```

- or

```
for (const Student &s: studs) cout << s << endl;
```

Range-based For Loops

- It is common practice to use **auto** in range-based for loops since the variable type will almost invariably be the type of the data items in the range:

```
for (auto &s: studs) cout << s << endl;
```

- There will be rare occasions when it is useful to use a variable whose type differs from the type of the data items when a conversion is available.
- One example could be using a **string** variable with a vector of C-strings to allow **string** member functions to be used:

```
vector<const char*> v(.....);
for (string s: v)
if (s.find('s')!=string::npos)
        cout << s << endl;
```

Range-based For Loops

- Consider an attempt to output the contents of a vector with position prefixes:

```
int i = 0;
for (int a: v3)
{
        cout << i << ':' << a << endl;
        i++;
}
```

- The traditional for loop would be more concise:

```
for (int i = 0; i<v3.size(); i++)
        cout << i << ':' << v3[i] << endl;
```

University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex