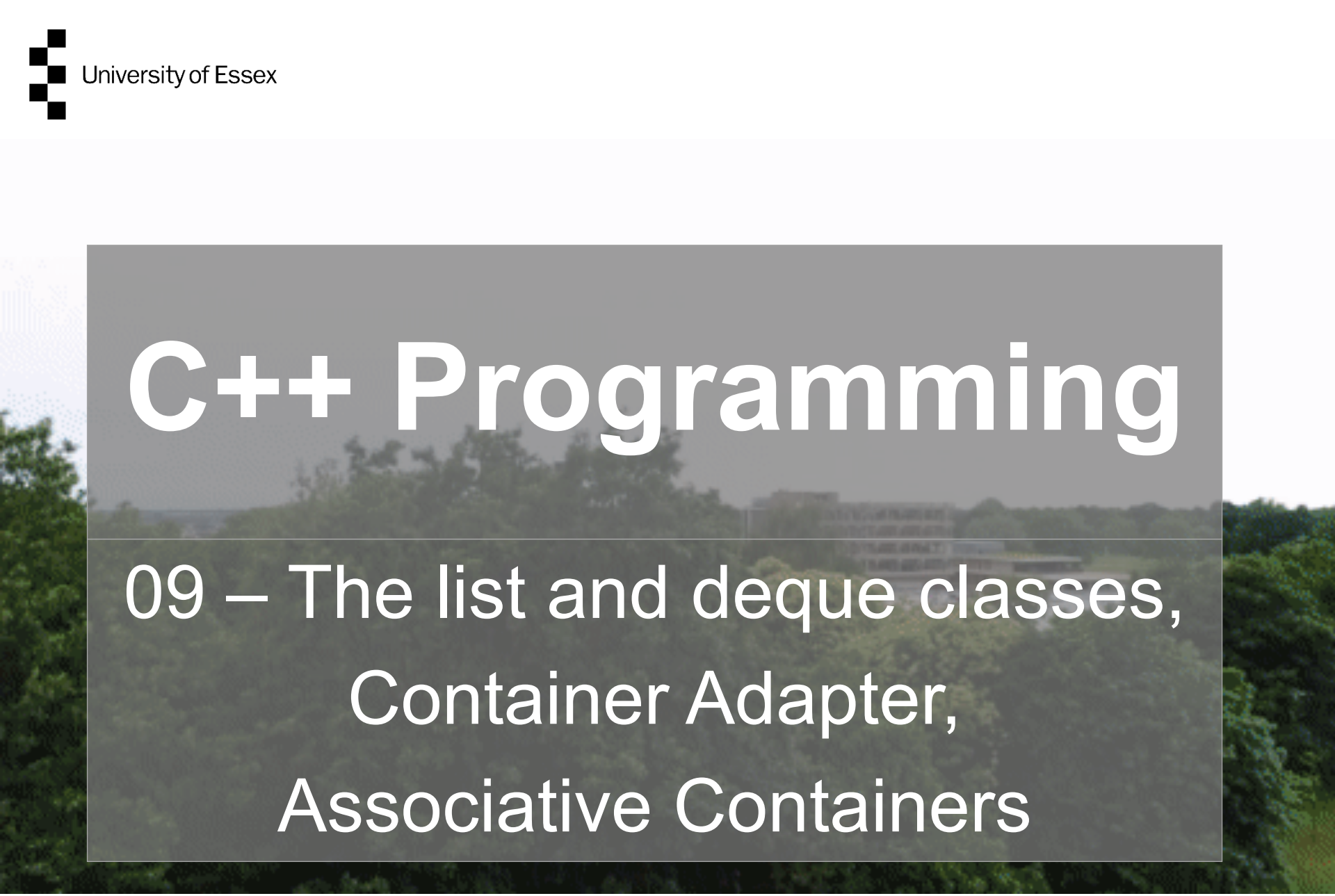
University of Essex
C++ Programming
09 – The list and deque classes,
Container Adapter,
Associative Containers

The list class

- The **list** container is a sequence container that stores the sequence as dynamically-allocated cells each containing a pointer to a data item and pointers to the previous and next in the list.
- However accessing items non-sequentially is not efficient so no subscripting or **at** functions are provided.
- Iterators for list objects are bidirectional – they support **++** and **--**, but not **+** and **+=**.

The list class

- The no-argument **list** constructor initialises the list to be empty.
- There is also a two-argument constructor – the first argument is a repetition count and the second a value: **list<int> l(3, 4)** will initialise **l** to hold 3 copies of the value 4.
- The first or last item of a list may be accessed using **front** or **back** and removed using **pop_front** or **pop_back**.
- There is no member function to remove items at other specified locations; to do this we need to use the iterator **erase** function.

The list class

- In addition to **push_back** the **list** class has a **push_front** member function for insertion at the front.
- There are **insert** methods that takes an iterator as the first argument; additional arguments are supplied in the same way as the methods from the **vector** class.

```
1  #include <list>
2
3  ....
4  list<int> l; //inside a main of course
5  l.push_back(3);
6  l.push_front(4);
7  l.insert(++l.begin(), 44);
8  for (int i:l) // assuming C++ 11
9    cout << i << " ";
10   cout << endl;
11           // output will be 4 44 3
```

The list class

- All occurrences of a particular value may be removed using remove function

```
1  list<int> l(4, 5); // 5 5 5 5
2  l.insert(++l.begin(), 7); // 5 7 5 5 5
3  list<int>::iterator pos = find(l.begin(),l.end(), 7);
4  l.insert(pos, 8); // 5 8 7 5 5 5
5  l.push_front(7); // 7 5 8 7 5 5 5
6  l.remove(7); // 5 8 5 5 5
7  cout << l.back(); // 5 will be output
8  l.pop_front(); // 8 5 5 5
```

The list class

- There is also a **remove_if** function that removes all values that satisfy a condition; the argument is a pointer to a function that takes a data item as its argument and returns a boolean value indicating whether that item satisfies the condition.
- To remove all upper-case letters from a list **l1** of type **list<char>** we could use **l1.remove_if(isupper)**.
- The following code will remove all negative values from the list **l2** of type **list<int>**.

```
1  bool isneg(int i)
2  {
3      return i<0;
4  }
5  l2.remove_if(isneg);
```

The list class

- We cannot use the sort algorithm to sort lists, since they do not support random-access iterators.
- Hence there are two **sort** member functions to sort a list into ascending order; a no- argument version that sorts using **<** and a version with one argument similar to the last argument of the three-argument **sort** algorithm.
- The function **unique** will remove from the list all but one of any sequence of adjacent equal items.
- If we want to eliminate all duplicate items from a list we need to sort it first so that equal items are adjacent.

The list class

- The **splice** function can be used to move all of the elements of one list into another.
- **l1.splice(pos, l2)** moves all of the elements of **l2** into **l1** immediately in front of the element referenced by the iterator **pos**.
- After this **l2** will be empty. Note that no copying of elements takes place – only the pointers in the list cells are changed.
- The function can take extra arguments to allow part of **l2** to be moved.
- The **merge** function can be used to move all of the elements of a sorted list into another, with the result being sorted.
- **s1.merge(s2)** moves all of the elements of **s2** into **s1** in appropriate positions to maintain correct ordering.
- If either of the lists is not sorted the order of elements is unpredictable.
- The functions can take an extra comparator argument to specify the ordering.

The deque class

- The **deque** container (pronounced as "deck" and being an abbreviation of double-ended queue) is similar to **vector** but supports efficient addition and removal at both ends.
- The **deque** class provides all of the functionality of **vector** and additionally **push_front**, **pop_front**, **front** and **back** functions.
- Programs that use this container need to include the header file **<deque>**.

The stack and queue adaptors

- The **stack** and **queue** container adaptors provide wrappers around sequence containers to allow them to be used as stacks and queues.
- A stack is a first-in-last-out sequence; items can be added to and removed from the top only and the only item that can be accessed directly is the top item.
- On the other hand a queue is a first-in-first-out sequence; items are added at the back but removed from the front and the only items that can be accessed directly are the front and back items.
- A program that uses one of these adaptors requires the use of **#include <stack>** or **#include <queue>**.

The stack class

- The **stack** class by default is implemented using a deque, but we may if we wish obtain a stack that is implemented using a vector or a list:

stack<int> s1; // uses deque<int>

stack<int, vector<int>> s2; // uses vector<int>

- We may initialise a stack to contain the contents of an existing sequence object of the same type as the class being used to implement the stack; the first item in the sequence will be at the bottom of the stack:

```
list<int> l;
.........
stack<int, list<int>> s3(l);
```

The stack class

- The main member functions of the **stack** class are **push**, **pop** and **top**. These are implemented using the functions **push_back**, **pop_back**, and **back**.
- **pop** and **top** have undefined behaviour if the stack is empty.
- In addition member functions **empty** and **size** (as described in part 7) are provided.

```
1  #include <stack>
2
3  ......
4  stack<int> st;
5  st.push(5);
6  st.push(4);
7  while (!st.empty())
8  {
9      cout << st.top() << endl;
10     st.pop();
11 }
```

University of Essex

The queue class

- The **queue** class by default also uses a deque, but we may if we wish obtain a queue that is implemented using a list (but not a vector since the **vector** class does not support **pop_front**):

```
#include <queue> queue<char> q1;
// uses deque<char>
queue<char, list<char>> q2;
// uses list<char>
```

- As with stacks we may initialise a queue to contain of the contents of an existing sequence object of the appropriate type; the first item in the sequence will be at the front of the queue.

The queue class

- The main member functions of the **queue** class are **push**, **pop**, **front** and **back**.
- The last three have undefined behaviour if the queue is empty.
- The **push** function adds to the back and the **pop** function removes the front element.
- Once again member functions **empty** and **size** are provided.

```
1  queue<char> q;
2  q.push('x');
3  q.push('y');
4  cout << "back is " << q.back() << endl; // y
5  while (!q.empty())
6  {
7      cout << q.front() << endl;
8      q.pop();
9  } // x will be output, then y
```

The priority_queue class

- A ***priority queue*** is like a queue but items have priorities and when an item is added to the back it will advance ahead of any items with lower priority.
- The order in which items with equal priority will reach the head of the queue not defined.
- The **priority_queue** class by default uses a vector, but we may obtain a queue that is implemented using a deque.

The priority_queue class

- Programs that use the **priority_queue** adaptor need to use

  #include <queue>.
- Priorities are by default compared using the **<** operator; in most applications the items in the queue will be class objects with the priority being a member of the class;
- we would need to write an **operator<** function for the class such that **a<b** returns true if **a** has lower priority than **b**.
- If the item class already has a **<** operator that compares something other than the priorities, we need to supply a comparator class;
- this class should be specified in the declaration of the queue, as a third template argument e.g.

  priority_queue<Job, deque<Job>, MyComp> pq;

The priority_queue class

- The comparator class needs to be a class with an **operator()** function that takes as arguments two data items **a** and **b** (or constant references to such items) and returns the value of **a<b** under the ordering to be used.
- For example if we wish to use a priority queue of objects of type **Job**, where **Job** is a class with a priority stored in a public instance variable **pri**, but has a **<** operator that does not compare priorities, we could write a class of the form

```
1  class MyComp
2  {
3      bool operator()(const Job &a, const Job &b)
4      {
5          return a.pri < b.pri;
6      }
7  }
```

The priority_queue class

- The member functions of the **priority_queue** class have the same names as those of the **stack** class.
- In the following example we assume that the **YellowPerson** class has a priority that is set using the last argument of its constructor, the **<** operator compares priorities and the **<<** operator has been overloaded for the class.

```
1  priority_queue<YellowPerson> pq;
2  pq.push(YellowPerson("homer", ......,2);
3  pq.push(YellowPerson("marge", ......, 3);
4  cout << pq.top(); // will be marge pq.pop();
5  pq.push(YellowPerson("bart", ......,2);
6  cout << pq.top(); // could be homer or bart
```

The pair Template Class

- C++ provides a template class called **pair** for storing pairs of objects. e.g.

```
pair<int, bool> p(3, false);
```

- We can access the elements of such a pair using the public data members **first** and **second**:

```
cout << p.first << ' ' << p.second << endl;
```

- If we wanted to supply a **pair** object as an argument to a function it would be inconvenient to specify its type explicitly so a template function called **make_pair** is provided

```
myfun(make_pair("bart", 45));
```

Associative Containers

- An associative container is used to store non-sequential data.
- The STL provides four such containers: **set**, **multiset**, **map** and **multimap**.
- In addition to the member functions listed in part 6 that are common to all containers, the associative containers all have member functions called **find**, **count**, **insert**, **lower_bound** and **upper_bound**.

Associative Containers

- Associative containers have bi-directional iterators but not random-access iterators.
- The **find** method performs the same task as the **find** algorithm (but in a more efficient way), i.e. **c.find(x)** will give the same result as **find(c.begin(), c.end(), x)**.
- In the case of a multi-map or multi-set the iterator that is returned will reference the first occurrence in the underlying implementation;
- We can increment the iterator to reach other occurrences.
- The **count** method returns the number of occurrences of its argument; for a set or map this will always be 0 or 1.

Associative Containers

- The **insert** method will insert an element into the collection; in the case of a set or map it will leave the collection unchanged if the element is already present.
- This method returns a result of type **pair<*C*::iterator, bool>** (where ***C*** is the instantiated container class, e.g. **set<int>**).
- The second element indicates whether anything was inserted; the first element is an iterator object that refers to the inserted element or the existing element with the same value if the element was not inserted.

```
1  set<int> s;
2  ......
3  if (s.insert(7).second)
4      cout << "7 inserted";
5  else
6      cout << "7 already present";
```

Associative Containers

- The **lower_bound** and **upper_bound** methods can be used to obtain start and end iterators for traversing a range of elements from a container.
- **s.lower_bound(x)** will return an iterator referencing the first occurrence of an element greater than or equal to x whereas **s.upper_bound(y)** will return an iterator referencing the first occurrence of an element greater than **y**.
- Both will return an iterator equal to **s.end()** if no such element exists.
- To visit all of the elements in the range **x** to **y** inclusive we can use a loop of the form

```
1  C::iterator end = s.upper_bound(y);
2  for (C::iterator it = s.lower_bound(x); it != end; it++)
3      // process *it
```

The set class

- The **set** container is used to store sets – a set may not contain duplicate elements.
- It has a no-argument constructor that will initialise a set to be empty, a copy constructor and a two-argument constructor that will initialise the contents of the set using all or part of an array or container object.
- The arguments are either start and end iterators or pointers
- (If the array or container object contains duplicate values only one occurrence of each value will be stored in the set.)
- The **<set>** header file needs to be included in programs that use sets (or multi-sets).

The set class

```
1  #include <set>
2  #include <string>
3
4  string a[6] = { "a", "b", "c", "d", "e", "f" };
5  vector<string> ( { "g", "h", "e", "i", "c", "a", "*****", "+++" } ) v;
6  set<string> A(a, a+6); // load A from a
7  set<string> B(v.begin(), v.begin()+6);
8  // load B from first 6 elements of b
9  set<string> C;         // initially empty
```

The set class

- Inserting an element into a set does not invalidate any iterators that already refer to elements of the set and removing an element using the erase method does not invalidate any iterators that do not refer to the deleted element.
- Hence we could use the following code to remove all numbers greater than n from the set s (of type set<int>).

```
1  set<int>::iterator it = s.begin(), it2;
2  while (it != s.end())
3  {
4      it2 = it++;
5      // need to advance it before deleting element
6      if (*it2 > n)
7          s.erase(it2);
8  }
```

The set class

- The ***intersection*** of two sets **s1** and **s2** is a set containing all elements that occur in both **s1** and **s2**, whereas the ***union*** of **s1** and **s2** is a set containing all elements that occur in **s1** or **s2** (or both).
- There are STL algorithms called **set_intersection** and **set_union** to obtain the intersection and union of two sets.
- The first four arguments for each of these functions are iterators referencing the beginning and end of the two sets; a fifth argument is used to specify a set in which the result should be stored.
- This takes the form of a special sort of iterator called an ***inserter***.
- A function call of the form **inserter(s, s.begin())** is used to generate an inserter for a set **s**.
- (This function is declared in the **<iterator>** header file.)

The set class

- In the following code we assume that the sets **A**, **B** and **C** are as declared on slide 25.

```
1  set_intersection(A.begin(), A.end(),
2                   B.begin(), B.end(),
3                   inserter(C, C.begin())); // C contains "a", "c", "e"
4  C.clear();
5  set_union(A.begin(), A.end(), B.begin(), B.end(),
6            inserter(C, C.begin()));
7  // C contains "a", "b", "c", "d", "e", "f", "g", "h ", "i",
```

The set class

- There is also an algorithm called **set_difference**; this is used to obtain a set containing all of the elements that appear in the set specified by the first two arguments but do not appear in the set specified by the third and fourth arguments.

```
1  C.clear();
2  set_difference(A.begin(), A.end(), B.begin(), B.end(),
3                 inserter(C, C.begin()));
4                 // C contains "b", "d", "f"
5  C.clear();
6  set_difference(B.begin(), B.end(), A.begin(), A.end(),
7                 inserter(C, C.begin()));
8                 // C contains "g", "i", "h"
```

The multiset class

- The **multiset** class is used for collections of elements which may contain duplicates;
- its methods are the same as those for **set** but some behave slightly differently.
- For example, the **insert** function will always succeed so the second element of the pair returned by the function will always be **true**.
- The number of occurrences of an item in the union of two multisets is the larger of its occurrence counts from the two sets whereas in the intersection it is smaller of the two occurrence counts.

The map class

- A ***map*** is a collection of keys with associated values, where each key must be unique (but values may occur many times).
- The **map** container in C++ is similar to Python's **dict** type – maps can be viewed as associative arrays and the values associated with keys can be accessed using subscripting.
- Since the keys and values are normally of different types the **map** template class takes two type parameters, e.g.

map<string, int> age;

- Programs that use this class (or **multimap**) need to include the header file **<map>** .

The map class

- The insert method for the map class takes a pair as its argument – it is possible to write code such as

  pair<string, int> p("lisa", 14); age.insert(p)

- Or

  age.insert(pair<string, int>("bart", 15));

- but it is inconvenient to have to explicitly specify the type parameters. A more concise approach is to use the template function **make_pair** described on slide 18:

  age.insert(make_pair("homer", 55));

- The insert method will not insert a pair if the key is already present in the map;
- it cannot be used to update the value associated with an existing key.

The map class

- An iterator for a map will reference a pair so we could print the contents of the **age** map using

```
1  map<string, int>::const_iterator it;
2  for (it = age.begin(); it != age.end(); it++)
3      cout << it->first << ":"<< it->second << endl;
```

- We could also use the **for_each** algorithm, supplying a function that takes a pair as an argument:

```
1  void printPair(const pair<string, int> &p)
2  {
3      cout << p.first << ":" << p.second << endl;
4  }
5  for_each(age.begin(), age.end(), printPair);
```

The map class

- The **find** and **count** methods for maps take a key as an argument:

```
1  map<string,int>::iterator it = age.find("bart");
2  if (it != age.end())
3      cout << "Bart's age is " << it->second << endl;
4  if (age.count("lisa") == 1)
5      cout << "Lisa is " << age.find("lisa")->second;
```

The map class

- As in Python we can access the value associated with a key using subscripting:

  int bartsAge = age["bart"];

- If the key is not present in the map the value of the subscript operation will be 0 (or equivalent) – no exception will be thrown.
- Subscripting is also used to update the value associated with a key :.

  age["maggie"] = 1; age["bart"]++;

- If the key is not present in the map a new pair will be added.

The multimap class

- In a ***multi-map*** a key may appear more than once. The methods for the **multimap** class are the same as those for **map**, apart from the absence of the subscripting operator, which cannot be supported since a key does not necessarily have a unique value.
- To retrieve the value(s) associated with a key we have to use the **find** method.
- Since this returns a reference to the first occurrence of the key we can increment the iterator to find subsequent occurrences.

The multimap class

- Assuming **tel** is a multi-map for the storage of telephone numbers we could use the following code to print all of Bart's telephone numbers:

```
1  multimap<string,int>::iterator it = tel.find("bart");
2  while (it != tel.end() && it->first == "bart")
3  {
4      cout << it->second << endl;
5      it++;
6  }
```

- Note that in the underlying implementation only the keys are sorted so we cannot determine in which order the values will be printed.

The multimap class

- To update a value associated with a key we must locate the pair using an iterator and then change the second element of the pair.
- Assuming one of Bart's numbers needs to be changed from 1234 to 5678 we might use

```
1  multimap<string,int>::iterator it = tel.find("bart");
2  while (it != tel.end() && it->first == "bart")
3  {
4      if (it->second == 1234)
5          { it->second = 5678;
6              break;
7          }
8      it++;
9  }
```

University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex