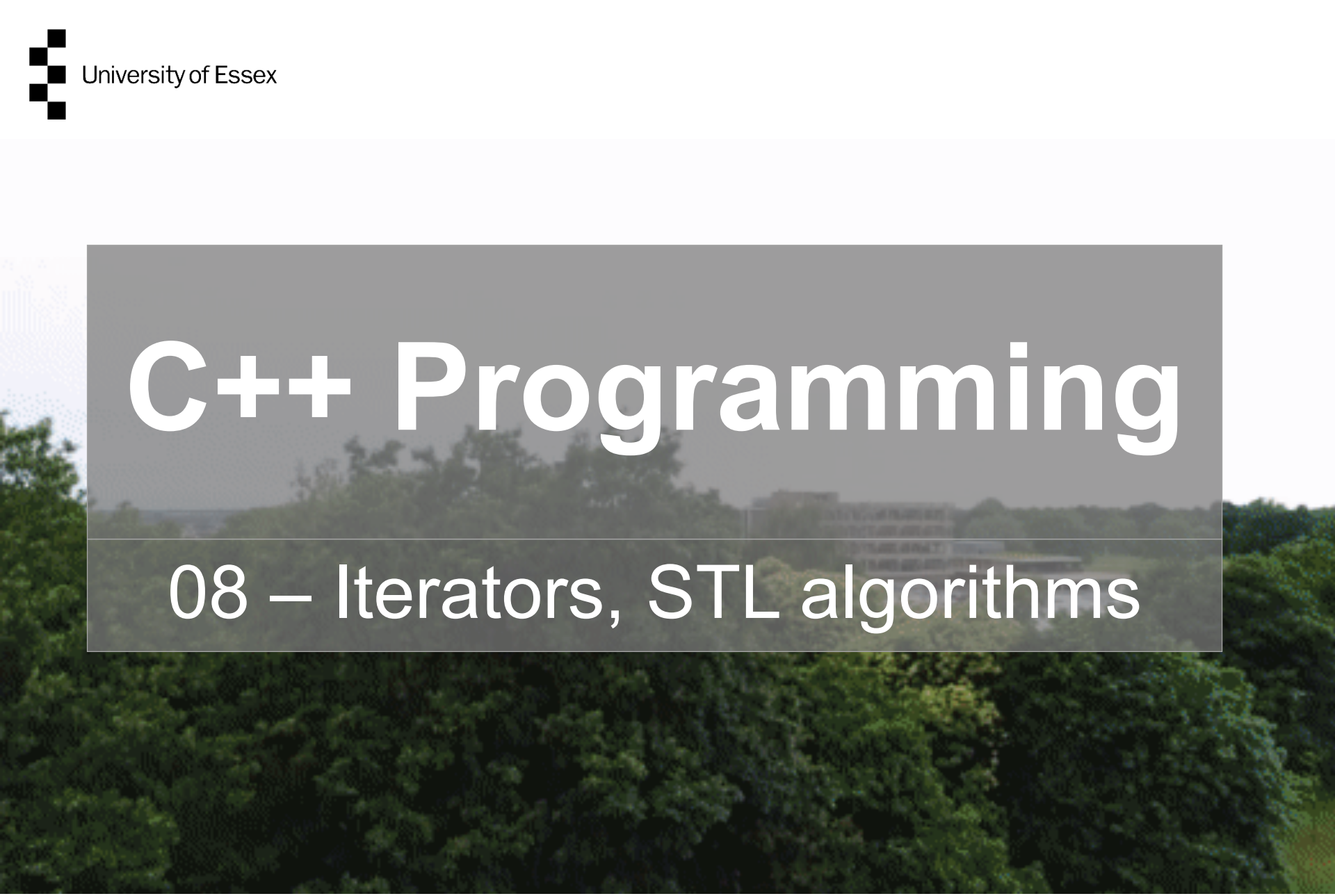
University of Essex



C++ Programming

08 – Iterators, STL algorithms

Iterators

- An ***iterator*** can be regarded as a smart pointer that points to each element in a collection in turn;
- Programs that use iterators should contain the line **#include <iterator>**.
- An iterator is not actually a pointer but the unary * operator has been overloaded so it can be used to refer to the element of the collection that the iterator “points” to.
- In addition the **++** operator has been overloaded to allow the pointer to be advanced to the next element in the collection.
- Some iterators also have overloaded version of operators such as **--** and **+=**.
- An iterator is declared using syntax such as **vector<int>::iterator it**. (The more natural **iterator<vector<int>>** cannot be used since iterators cannot be implemented using a template class.)

Iterators

- There are three different types of iterator: unidirectional (with just **++**), bidirectional (with **++** and **--**) and random-access (with full "pointer" arithmetic);
- To obtain an iterator that starts at the beginning of a collection the **begin** function from the container class should be used, e.g.

vector<int>::const_iterator it = myvec.begin();

- The function returns an object of type ***T*::iterator** where ***T*** is the type of the object to which it is applied.

Iterators

- The **end** function from the container class returns an iterator whose pointer position is just after the last element of the collection so we can compare the current state of our iterator with **myvec.end()** to determine whether all items have been traversed.
- We should use **==** or **!=** to perform the comparison since not all iterators have **<** or **<=** operators.
- The **end** function also returns a non-constant iterator, but the **==** and **!=** operators regard a constant iterator and a non-constant iterator that point to the same element in a collection as being equal.

Iterators

- We can use the following loop to print the contents of a vector **v** with element type **int** one item per line.

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

- The following loop will replace all values greater than 100 in the vector with the value 0. (This time we cannot use a constant iterator.)

```
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); it++)
        If (*it > 100)
                *it = 0;
```

Iterators

- If we wish to traverse a collection in reverse order and the container supports only unidirectional iterators we need to use a ***reverse iterator***.
- The following loop will print the contents of **v** in reverse order.

```
vector<int>::const_reverse_iterator rit;
for (rit = v.rbegin(); rit != v.rend(); rit++)
        cout << *rit << endl;
```

- Observe that we need to use the functions **rbegin** and **rend** to obtain reverse iterators, and **++** (not **--**) is used to move to the previous element.

Iterators

- The **vector** and **string** classes support random-access iterators so it is not in fact necessary to use a reverse iterator to access the contents in reverse order; we could use **--** to step through the elements from **v.end()** to **v.begin()**.
- Since we need to access the element referred to by **v.begin()** but not the one referred to by **v.end()** we cannot use a for loop with a similar structure to the previous slides.

```
vector<int>::const_iterator it = v.end();
while (it != v.begin())
{
        it--;
        cout << *it << endl;
}
```

The erase Function

- The **erase** function of a container class may be used to remove one or more elements from a collection.
- The single-argument version simply removes the element currently pointed to by the iterator; an exception will be thrown if the iterator does not point to an element of the collection.
- We can remove the first element from a vector **v** (or any other container) using **v.erase(v.begin())**.
- To remove the element at location **n** we could use **v.erase(v.begin()+n)**; this technique can be used only with containers that have random- access iterators.

The erase Function

- We could use the following code to locate and remove the first occurrence of 0 in the vector **v**.

```
vector<int>::iterator it = v.begin();
while (it != v.end() && *it != 0)
        it++;
if (it != v.end())
        v.erase(it);
```

The erase Function

- The two-argument version of **erase** will remove a sequence of elements starting with the element pointed to by the first argument and ending immediately before the element pointed to by the second argument so to remove the elements at locations 3, 4 and 5 from the vector **v** we could use **v.erase(v.begin()+3, v.begin()+6)**.
- and to remove all but the first six elements we could use **v.erase(v.begin()+6, v.end())**.
- The two arguments must have the same type so we cannot use a combination of a normal iterator and a reverse iterator.
- An exception will be thrown if the iterator does not point to an element in the container, e.g. if we try to use something like **v1.erase(v2.begin())**.

Inserting into Vectors

- The **vector** class has three **insert** member functions allowing new elements to be inserted at a position specified by an iterator.
- In all of these, the element(s) will be inserted in front of the position pointed to be the first argument;
- An exception will be thrown if the iterator does not "point" either to an element of the collection or to the position immediately after the last element.

Inserting into Vectors

- The simplest **insert** function has just two arguments, the second being the item to be inserted.

```
v.insert(v.begin()+2, 4);
// inserts 4 in front of v[2]
```

- The second **insert** function allows multiple copies of the same element to be inserted.
- This takes the number of copies as its second argument and the element as its third argument.

```
v.insert(v.begin()+3, 2, 5);
// inserts 2 copies of 5 in front of v[3]
```

Inserting into Vectors

- The third **insert** function allows a sequence of elements from another collection (which does not have to be a vector) or from an array to be inserted at a position specified by an iterator.
- The second argument specifies the location of the beginning of the sequence and the third the location immediately after the end of the sequence.

```
int a[] = .........
v.insert(v.begin(), a, a+5);
// inserts copies of first 5 elements of a
```

typedef

- If several functions in a program used constant reverse iterators for **vector<int>** it would be inconvenient have to use **vector<int>::const_reverse_iterator** in many declarations.
- To make the code more concise we can make use of **typedef**.
- After defining the type **CRI** using

  typedef vector<int>::const_reverse_iterator CRI;

- we could declare constant reverse iterators by simply using declarations of the form **CRI it;**.

typedef

- The position of the name of a type in a **typedef** statement is always the same position as that of a variable of that type in a declaration.
- Hence, for example, since **int *p[40];** would define **p** to be an array of 40 pointers to integers,
- **typedef int *PA[40];** would define the type **PA** to be "array of 40 pointers to integers".
- [We can tell that the variable declaration gives an array of pointers rather than a pointer to an array since the precedence rules state that the expression ***p[39]** means ***(p[39])** so **p** must be an array and **p[39]** must be a pointer.]

typedef

- It is possible to include **typedef** statements inside class declarations. Consider for example

```
class C
{
        typedef ...X...
        ......
}
```

- Inside the class declaration we can use the name **X** to refer to the type specified by the **typedef** statement; outside the class we would have to use **C::X**.
- This is in fact how the writers of the standard template library were able to define types like **vector<int>::iterator**.

Using auto in C++11

- C++ 11 introduced a facility for the compiler to infer types of variables from their initialisation, avoiding the need to explicitly use lengthy type names in declarations.
- In a declaration of the form

  auto x = *e*;

- the type of **x** will be the type of the expression ***e***.
- Note that if ***e*** is a call to a function that returns a reference the type of **x** will ***not*** be a reference;
- if we want a reference we must explicitly use the **&** symbol:

  auto &x = myfun(y);

Using auto in C++11

- Using C++ 11 we could have written on slide 7

  auto it = v.end();

- instead of

  vector<int>::const_iterator it = v.end();

- The inferred type of **it** would be **vector<int>::iterator**, but we may be willing to accept this if we know that our code will not change the contents of **v**, and do not require the compiler to check this.

Using auto in C++11

- Note that **auto** can be used only when the variable is given an
- initial value in its declaration. We could not replace

```
vector<int>::const_reverse_iterator rit;
```

- from slide 6 with

```
auto rit;
```

- However, we could have used

```
auto rit = v.begin();
for (; rit != v.rend(); rit++)
        cout << *rit << endl;
```

STL Algorithms

- The standard template library has a large collection of functions known as ***algorithms*** that can be used to search and manipulate the contents of containers or strings.
- The STL has about 70 algorithms, including **find**, **replace**, **search**, **sort**, **copy**, and **remove**.
- Not all algorithms work with all container classes; for example a set is an unordered collection of items ({1, 2, 3} is the same set as {3, 1 ,2}) so there is no concept of being sorted for a set.

STL Algorithms

- All algorithms take two iterator objects as arguments, specifying the start and end of the portion of the collection to which the algorithm is applied.
- (If this is the whole collection the results returned by **begin** and **end** or **rbegin** and **rend** should be used.)
- There will often be additional arguments.
- As usual the end argument should be an iterator that references the element after the last item in the portion.
- To use most algorithms a program must contain the line **#include <algorithm>**.
- However there are some numerical algorithms (which work only with collections of numbers or objects that have appropriate operators, e.g. **operator+**) for which the line **#include <numeric>** should be used instead.

find

- The **find** algorithm will return an iterator that references some occurrence of a specific value, supplied as a third argument.
- In the case of a sequence container or string this will be the first occurrence (or last occurrence if a reverse iterator is being used).
- If the value does not occur in the collection the end iterator for the collection or portion of the collection (i.e. that supplied as the second argument) will be returned.
- The code fragment on the next slide will print the contents of the vector **v** (of items of type **int**) from the first occurrence of 0 to the end of the vector and also the substring of the string **s** extending from the character following the first occurrence of **'*'** to the end of the string.

find

```
1  typedef vector<int>::const_iterator VecIter;
2  typedef string::const_iterator StrIter;
3  VecIter zero = find(v.begin(), v.end(), 0)
4  for (VecIter it1 = zero; it1!=v.end(); it1++)
5      cout << *it1 << ' ';
6  cout << endl;
7  StrIter star = find(s.begin(), s.end(), '*')
8  if (star != s.end())
9      for (StrIter it2 = star+1; it2!=s.end(); it2++)
10         cout << *it2;
11 cout << endl;
```

find

- The following function will return the number of occurrences of the character **c** in the string **s**

```
1  int count(char c, const string& s)
2  {
3      int occs = 0;
4      string::const_iterator it = find(s.begin(), s.end(), c);
5      while (it!=s.end())
6          {
7              occs++;
8              it = find(it+1, s.end(), c);
9          }
10     return occs;
11 }
```

find

- We could also write a template version that will count the number of occurrences of an item in any collection with the correct item type.

```
1  template <class T, class C>
2  int count(T val, const C& s)
3  {
4      int occs = 0;
5      typename C::const_iterator it = find(s.begin(), s.end(), val);
6      while (it!=s.end())
7      {
8          occs++;
9          it = find(it+1, s.end(), val)
10     }
11     return occs;
12 }
```

find

- There is nothing in the code on the previous slide to indicate that **C** should denote a container class and the objects in **C** should have type **T**.
- However if an attempt were made to call the function with classes that do not satisfy this a type error would be raised when trying to generate a call to **find**.
- Since the compiler does not know that **C** should denote a container class it would not know that **C::const_iterator** is a type rather than a class member and would try to treat it as the latter unless it is informed that it is the name of a type; hence we have to use the keyword **typename**.

sort

- The two-argument **sort** algorithm will sort the contents of a collection (or part of a collection) into ascending order (i.e. smallest first).
- It can be used only with containers that support random-access iterators.
- Comparison of items within the function is performed using the **<** operator so if the items in the collection are objects they must belong to a class that has an **operator<** function.
- The function changes the order of the contents within the container so it does not make a copy and does not return a result.
- If the arguments are reverse iterators the contents of the collection will be sorted into descending order.

sort

```
1  #include <vector>
2  #include <iterator>
3  #include <algorithm>
4  #include <iostream>
5
6  using namespace std;
7
8  template <class T>
9  void printvec(vector<T> v)
10 {
11     cout << '<';
12     vector<T>::const_iterator it = v.begin();
13     while (it!=v.end()) {
14         cout << *it;
15         if (++it != v.end()) cout << ',';
16     }
17     cout << '>' << endl;
18 }
19
20 int main()
21 {
22     vector<int> v;
23     v.push_back(7);
24     v.push_back(17);
25     v.push_back(3); // v now holds [7,17,3]
26     vector<int> v2(v); // v2 also holds [7,17,3]
27     sort(v.begin(), v.end());
28     printvec(v); // outputs <3,7,17>
29     sort(v2.rbegin(), v2.rend());
30     printvec(v2);       // outputs <17,7,3>
31 }
```

sort

- There is a three-argument **sort** algorithm that will sort the contents using a programmer-supplied comparison function instead of **<**.
- This is useful when we wish to sort a sequence of objects and the value to be used for sorting is one of the members of the class but either the **operator<** function of the class compares a different member or the class has no **operator<** function.
- The third argument to **sort** should be a pointer to a boolean function that takes two arguments (constant references to the items to be compared) and returns true if the value of the first argument is less than the value of the second argument using the ordering that we wish to use.

sort

- The code below will sort a vector **v** of type **Vector<student>** by descending average mark (assuming that **aveMark** is a public member of the **Student** class).
- [Recall that to pass a pointer to a function as an argument to another function we simply supply the function's name.]

```
1  bool compareMark(const Student& s1, const Student& s2)
2  {
3      return s1.aveMark < s2.aveMark;
4  }
5  // need a reverse iterator for descending order
6  sort(v.rbegin(), v.rend(), compareMark);
```

for_each

- The **for_each** algorithm provides in C++ some of the functionality of Java's **for int i:myList** or Python's **for i in myList**.
- Its use is not required since the introduction of range-based for loops in C++11.
- However a programmer may have to adapt code that was written before C++11 so should know how to use it.
- As **for_each** is a function rather than a loop construct we have to provide the "loop body" as an argument.

for_each

- We can print the contents of a vector of integers one item per line by supplying a function that will print a single integer and a newline as an argument to **for_each**:

```
1  void print(int i)
2  {
3      cout << i << endl;
4  }
5  vector<int> v;
6  .........
7  for_each(v.begin(), v.end(), print);
```

for_each

- Since the "loop body" in C++ has to be written as a function it cannot access any local variables from the function that makes the call to **for_each**

```
 1  int sum;
 2  void addtoSum(int i)
 3  {
 4      sum += i;
 5  }
 6
 7  int main()
 8  {
 9      .......
10      sum = 0;
11      for_each(v.begin(), v.end(), addtoSum);
12  }
```

accumulate

- There are ways to overcome the need to use a global variable by using class objects but these are complicated.
- Since the task of obtaining value such as sums, products, minimum and maximum values is often needed and could not be done easily nefore the introduction of range-based for loops an algorithm to perform such tasks is provided in the STL. It is called **accumulate** (and declared in the **<numeric>** header.)
- We could obtain the sum of the items in a vector using

  sum = accumulate(v.begin(), v.end(), 0)

- The third argument is a starting value. Items from the collection are repeatedly added to this with the final value being returned.

accumulate

- Note that the return type of **accumulate** is determined by the type of the third argument since it is being used to build up the result. If we used

sum = accumulate(v.begin(), v.end(), 0.0)

- the sum would be a real number.
- To use **accumulate** to generate anything but the sum we need to supply as a fourth argument a pointer to a two-argument function to be performed instead of addition
- Its first argument and return type need to have the same type as the third argument to **accumulate** and the type of its second argument should be the type of the items in the collection.

accumulate

- A call to **accumulate(it, end, x, f)** will effectively perform a loop of the form

```
while (it != end)
        x = f(x, *it++);
```

- and return the final value of **x**.
- To use **accumulate** to find the minimum value in a collection of integers we could write a function

```
int minimum(int m, int n) { return m < n ? m : n; }
```

- The first argument represents the smallest item found so far and the second represents the current item in the collection; the value returned is the updated smallest item found so far.

accumulate

- The initial value for the smallest item so far needs to be supplied as the third argument to **accumulate** – we should use the largest **int** value supported by the C++ implementation
- This can be obtained using **numeric_limits<int>::max()** (declared in the header file **<limits>**).
- Assuming our program contains the **minimum** function on the previous slide, a call to **accumulate** to find the minimum value in a vector **v** of integers should be of the form

```
min = accumulate(v.begin(), v.end(),
                 numeric_limits<int>::max(),
                 minimum);
```

accumulate

- Suppose we have a collection of objects of type **Student** with a member called **mark** (of type **float**) and wish to find the highest mark in the collection.
- This time we need to write a function which extracts the mark from the current object and compares it with the maximum so far.
- Assuming no student can have a negative mark the appropriate start value for the maximum so far is 0.

```
float maxMark(float m, const Student &s)
{
        return s.mark > m ? s.mark : m;
}
.........
max = accumulate(studs.begin(), studs.end(), 0.0, maxMark);
```

University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex