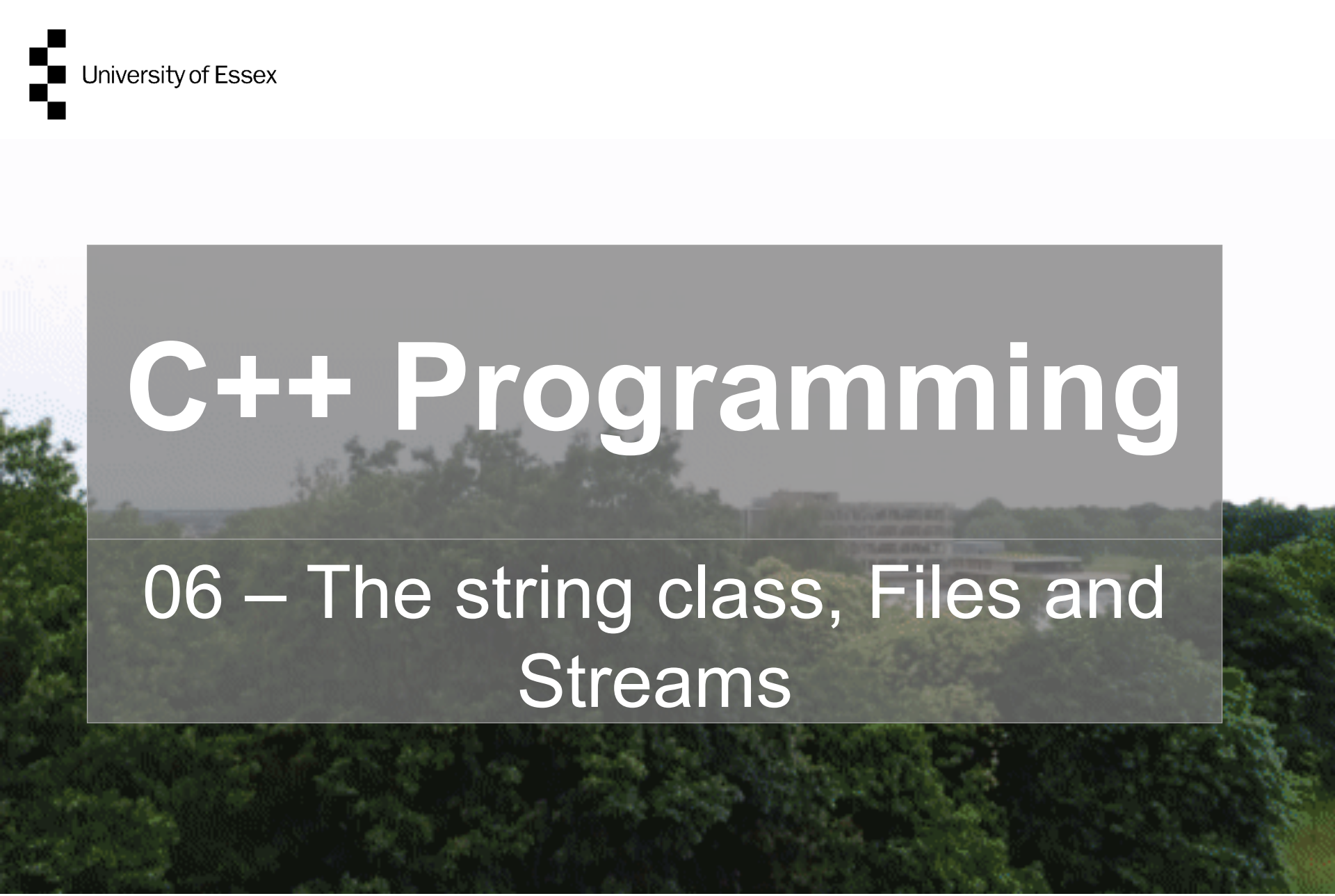
University of Essex



C++ Programming

06 – The string class, Files and Streams

The string Class

- C++ provides two ways of storing strings. They may be stored as C-strings in arrays using a terminator character **'\0'** to denote the end of a string or as objects of type **string**.
- A string literal, such as **"Hello**", is treated as a character array, the string being represented as a C-string.
- For simple string usage C-strings are normally more efficient than string objects;
- However, for robust programming and complex string manipulation the **string** class is more useful.
- Any file that uses the **string** class must include the appropriate header file:

#include <string>

The string Class

- The string class has several constructors.
- There is a constructor with no arguments which initialises a string to be empty,

```
#include <string>
string s1; // initially empty
string s2("Hello"); // initially holds Hello
string s3(s2); // initially a copy of s2
```

- There is also a constructor with arguments of type **int** and **char**:

```
string s4(5,'x'); // initially holds xxxxx
```

The string Class

- The constructor with a **const char*** argument can be used as a type conversion constructor
- There is no constructor with just a character argument
- We cannot write a declaration such as **string s('a');** (although we could use **string s(1, 'a');** .
- In addition to the assignment operator there are overloaded versions of **operator=** with argument types of **const char*** and **char** so assignments such as **s1 = "xx"** and **s1 = 'a'** are permitted.

The string Class

- Characters in a string object can be accessed using a member function called **at** or the **[]** operator which has been overloaded.

```
1  string s("Hello");
2  cout << s.at(1); // outputs e
3  s.at(2) = 'e'; // s now holds Heelo
4  cout << s[4]; // outputs o
5  s[4] = 's' // s now holds Heels
```

- The **at** function will throw an exception if its argument is out of range but the **[]** operator does not perform range-checking so its use can lead to runtime errors.

The string Class

- The **string** class has many functions and overloaded operators.
- We can use the **append** function or **+=** to append to the end of a string a copy of another string or a C-string:

```
string s1("Hello"), s2(s1);
s1 += " world"; // now holds Hello world
s2.append(s2); // now holds HelloHello
```

- The second operand of **+=** may also be a character and there is a version of the **append** function which appends multiple copies of a character:

```
s1 += '!'; s2.append(3, '!');
```

The string Class

- There is another version of the **append** function which appends a copy of part of another string: **s2.append(s1, 3, 2);**
- The **+** operator may be used to concatenate strings.

```
s3 = s1+s2;
s4 = s3+'!';
s5 = "***"+s4;
```

The string Class

- The **insert** function may be used to insert content at any place in a string;
- **s1.insert(m, x)** will insert a copy of **x** (which may be a **string** object or a C-string) into **s1** in front of the character at location **m**.
- As with **append** there is a version with a **char** parameter that takes an extra number-of-copies argument:

```
String s("C21");
s.insert(1, "E2"); // now holds CE221
s.insert(5, 2, '+'); // now holds CE221++
```

The string Class

- The length of a string may be obtained using either of the **length** and **size** member functions.
- The **substr** function may be used to extract a substring from a string.
- It is a member function with two arguments: **s.substr(m, n)** will generate a substring starting of length **n** with the character at location **m**.

The string Class

- We can replace part of a string using the **replace** function: **s1.replace(m, n, x)** will replace the substring of length **n** starting at location **m** with a copy of **x**, which may be a character, another **string** object or a C-string.
- To delete a substring from a string we can use the **erase** function which takes two integer arguments analogous to those of **substr**.

```
String s("Hello");
s.replace(2, 2, "xxy"); // now holds Hexxyo
s.erase(1, 3) // now holds Hyo
// could also have used s.replace(1, 3, "")
```

The string Class

- Strings may be compared using the six comparison operators.
- The **<**, **<=**, **>** and **>=** operators compare strings lexicographically using the ASCII ordering.
- The six operators also allow a **string** object to be compared with a C-string.
- There is also a **compare** function: **s1.compare(s2)** returns 0 if the strings are equal, a negative number if **s1** lexicographically precedes **s2** and a positive number otherwise.
- There are also several other versions of **compare** with extra arguments that allow comparisons of substrings.

The string Class

- There are several member functions for searching in strings.
- If the item was not found the special value **string::npos** is returned.
- A second argument may be supplied, indicating the position in the string where searching should start.
- The **find** function when used with one argument will attempt to locate the first occurrence in the string; the **rfind** function will attempt to locate the last occurrence.

The string Class

- If the string **s** held **Hello** then **s.find('l')** would return the value 2 but **s.rfind('l')** would return 3.
- **s.find('l', 3)** would search for the first occurrence of **l** at or after location 3 and hence return 3, and **s.rfind('l', 2)** would search for the last occurrence of **l** at or before location 2 and hence return 2.

The string Class

```
1  using namespace std;
2
3  #include <iostream>
4  #include <string>
5
6  int countOccs(string s, char c)
7  {
8      int occs = 0;
9      int pos = s.find(c);
10     while (pos!=string::npos)
11     {
12         occs++;
13         pos = s.find(c, pos+1);
14     }
15     return occs;
16 }
17
18 int main
19 {
20     String s("abcabcabcab");
21     cout << countoccs(s, 'b'); // should output 4
22     cout << countoccs(s, 'd'); // should output 0
23 }
```

The string Class

- The functions **find_first_of** and **find_first_not_of** can be used to find the first occurrence of any character from an argument string.
- For example we could use something like **s.find_first_not_of(" \t\n")** to find the first non-white- space character in a string.
- There are also functions **find_last_of** and **find_last_not_of**.

Inputting Strings

- Use of the overloaded **>>** operator to input a **string** object will cause the next sequence of non-white-space characters to be read from the input stream, so it cannot be used to input strings that contain spaces.
- To input a line of text that may contain white space into a string we need to use the function **getline**.
- This takes two arguments, a reference to the input stream and a reference to a **string** object in which the line is to be stored. e.g. **getline(cin, s2)**.
- The newline character at the end of the line will be consumed but will not be stored in the **string** object.

String Streams

- A ***string stream*** allows “input” and “output” to be performed from and to **string** objects.
- This allows us to use the stream formatting facilities such as **setw** and **setfill** provided when generating strings from data and to convert data in strings into objects of other types.
- An output string stream can be created using the no-argument constructor of the **ostringstream** class; this will create a string object to store the “output”.
- We can then use the **str** function from the class to access the string.
- We could use an output string stream to write an **operator string** function for our **Time** class (this would be declared in the class as **operator string() const**).

String Streams

```
1  using namespace std;
2
3  #include <iostream>
4  #include <string>
5  #include <sstream> // for string streams #include "Time.h"
6
7  Time::operator string() const
8  {
9      ostringstream s;
10     s << setfill('0') << setw(2) << hour << ":" << setw(2)
11     << min << ":" << setw(2) << sec;
12     return s.str();
13 }
```

String Streams

- An input string stream is created by supplying a **string** object as an argument to the constructor of the **istringstream** class.
- We can then extract data items from the string using the **>>** operator.
- We could use this to extract numeric data from a string containing digits; assuming that **s** is such a string we could write code such as

```
istringstream str(s);
int i;
str >> i;
```

- We can check whether the extraction was successful by applying the **good** function to the stream, e.g.

```
if (!str.good())
        cout << "error reading from string";
```

File Processing

- Programs may access files for input and output using file streams.
- The class **ifstream** inherits from **istream** so we can use **>>** to obtain input from a file; and similarly the class **ofstream** inherits from **ostream** so we can use **<<** to output to a file.
- A file stream object may be associated with a file using its constructor (e.g. **ifstream str("data.txt");**) or using the open function e.g.

```
ofstream str;
str.open("output.txt");
```

- Programs that use file streams need **#include <fstream>**.

File Processing

- The **!** operator has been overloaded for file streams so **!str** will be true if the file could not be opened.
- The first argument to file stream constructors and the **open** function must be a C-string; they can take a second argument which specifies an ***access mode*** (if this is not supplied it defaults to **ios::in** for input streams and **ios::out** for output streams).
- When **ios::out** is used for an output stream any contents of the file (if it already exists) will be overwritten. To append output to the end of an existing file **ios::app** should be used instead:

```
ofstream str;
str.open("output.txt", ios::app);
```

File Processing

- Streams may be associated with binary files; in this case the appropriate access mode (for both input and output) is **ios::binary**.
- A file will be closed by the destructor when the lifetime of the file stream object is about to expire; we sometimes need to close files before this happens,
- A file associated with a stream **str** may be explicitly closed using **str.close()**; after doing this any attempts to perform input or output on that file will fail.

Input Stream Functions

- As an alternative to the use of **>>** we can obtain input from an input stream using member functions.
- The **get** function will return the next character in the file; this may be white space.
- If the end of the file has been reached the special value **EOF** would be returned.
- To process the characters from a stream one by one we could use a loop of the form

```
char c;
while ((c = str.get()) != EOF) // process c
```

- It is also possible to check whether the end of a file has been reached using **str.eof()**, which returns a boolean result.

Input Stream Functions

- It is also possible to input the contents of a file line by line, using the **getline** member function of the **istream** class.
- This takes two arguments, the address of the beginning of a character array which will be used to store the line as a C-string, and an integer indicating a maximum size of the string.
- (This function is not the same as the **getline** function described on slide 16; that is used to input a line into a **string** objects and is not a member of the **istream** class.)

Input Stream Functions

- Consider the following code.

```
1  ifstream str("data.txt");
2  const int SIZE = 80;
3  char buffer[SIZE];
4  while (!str.eof())
5  {
6      str.getline(buffer, SIZE);
7      // process buffer
8  }
9
10 // using namespace std;
```

- If a line in the input file contained more than 79 characters only the first 79 would be input (the size argument includes space for the end-of-string delimiter); the rest of the line would remain in the input stream to be read by the next call to the function.

Formatting Output

- It is possible to output to a file one character at a time using the **put** member function; this is rarely used since **str.put(c)** is equivalent to **str << c**.
- The **width** function can be used to specify a minimum field width for the next item to be output; if the item requires less space fill characters are inserted as padding (the default fill character is a space).
- To output an number **n** in a field of width 6 we could use **cout.width(6); cout << n;**

Formatting Output

- The **fill** function may be used to change the fill character.
- To display a number in a fixed-with field with leading zeroes instead of spaces we could use

```
cout.fill('0’);
cout.width(6);
cout << n;
cout.fill(' ');
```

- The last line would of course not be necessary if the next item to be output in a fixed-width field was another number where leading zeroes are required.

Formatting Output

- Real numbers may be output in fixed-point (e.g. **0.34**) or scientific format (e.g. **3.4e-01**).
- To specify a particular format for use for all subsequent real number outputs we should use the **setf** function with an argument of **ios::fixed** or **ios::scientific**.
- We can specify a precision using the **precision** function.

I/O Manipulators

- It is usually more concise to place format control inside a chain of **<<** operations.
- This can be done using ***i/o manipulators***, e.g.

cout << setfill('0') << setw(5) << n << setfill(' ') << endl;

- The **setw** and **setfill** manipulators perform the same tasks as the **width** and **fill** functions.
- There is also a **setprecision** manipulator.
- When using i/o manipulators it is necessary to use the line

#include <iomanip>

I/O Manipulators

- The manipulators for selecting the format for real-number output are simply **fixed** and **scientific**.
- These are used without parentheses and are not preceded by **ios::**.

```
cout << fixed << setprecision(5) << pi << endl;
```

- We can also select left- or right- alignment for items in fixed- width fields using **left** or **right**.

```
cout << left << setw(5) << n << endl;
```

- (It is also possible to set alignment using functions but these are more complicated than the examples we have seen and hence rarely used.)

University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex