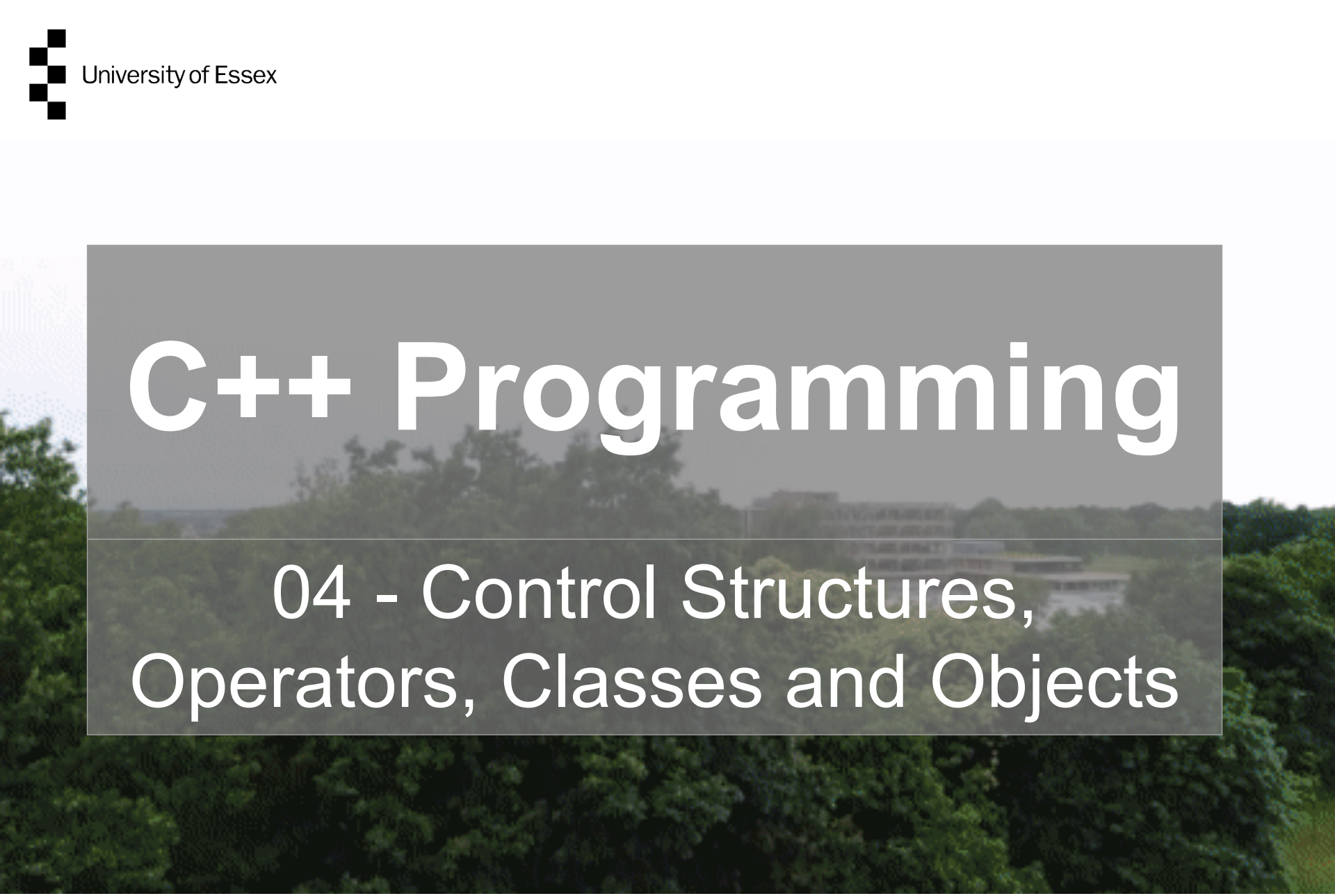
University of Essex

# C++ Programming

## 04 - Control Structures, Operators, Classes and Objects



# Control Structures

- The control structures in C++ are essentially the same as in Java.
- Selection structures: **if** (with optional **else**) and **switch**
- Repetition structures: **while**, **do** and **for**
- Exception handling: **try**/**catch**
- However the declaration of user-defined exception classes differs from Java. (Exception-handling in C++ will be covered later.)

# Control Structures

- The **break** and **continue** statements can be used to break out of control structures.
- The **break** statement causes the program to jump out of the nearest enclosing loop or **switch** statement
- The **continue** statement (which is used only in loops) causes to program to jump to the end of the current iteration
- It proceed by checking the condition for continuing in the loop

# break and continue – an Example

```
for (init; condition; update) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
```

```
for (init; condition; update) {
    // code
    if (condition to break) {
        continue;
    }
    // code
}
```

```
while (condition) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
```

```
while (condition) {
    // code
    if (condition to break) {
        continue;
    }
    // code
}
```

University of Essex

# Default Argument To Functions

- A function can have arguments with default values, e.g.

```
int volume(int length = 1, int width = 1,
           int height = 1) { ..... }
```

- **volume(3,4)** would be treated as **volume(3,4,1)**. (The supplied values are used for the first arguments.)
- If some but not all of the arguments of a function have default values they must appear at the end of the list, e.g.

```
void printArea(float length, float width, int
               decimalPlaces = 2) { ..... }
```

# Function Overloading

- It is permissible to write more than one function with the same name as long as they have different numbers of arguments or different argument types.
- The compiler will determine which version to use for each call by looking at the arguments.
- The return types of the overloaded functions may the same or may be different, e.g.

```
int square(int x);
float square(float x);
float area(float height, float width);
float area(float radius);
```

# More about Functions

- In C++, it is not possible to declare a function inside the body of another function.
- Short simple functions are sometimes declared as **inline**, e.g.

```
inline int cube(int i) { return i*i*i; }
```

# Operators and Precedence

- C++ has many operators.
- It has all of the operators of Java apart from **instanceof**.
- In addition there are the operators **->**, **::** and **delete** and the unary versions of * and **&** that we have already encountered, and there are **.***, **->***, several cast operators and **sizeof**.
- The operator precedence table is shown on the following slides, with the highest precedence first, so for example ***p.q** means ***(p.q)** .

# Operator Precedence Table 1

| Operators | Function | Associativity |
|---|---|---|
| *::* | binary scope resolution | left to right |
| *::* | unary scope resolution | |
| *()* | function call | left to right |
| *[]* | Array subscript | |
| . | Member selection via object | |
| -> | Member selection via pointer | |
| *++* | Unary postfix increment | |
| -- | Unary postfix decrement | |
| *typeid* | Runtime type information | |
| *dynamic_cast<type>* | Runtime type-checked cast | |
| *static_cast<type>* | Compile-time type-checked cast | |
| *reinterpret_cast<type>* | Cast for nonstandard conversions | |
| *const_cast<type>* | Cost away const-ness | |

# Operator Precedence Table 2

| Operators | Function | Associativity |
|---|---|---|
| *++* | Unary prefix increment | Right to left |
| -- | Unary prefix decrement | |
| *+* | Unary plus | |
| - | Unary minus | |
| *!* | Unary logical negation | |
| ~ | Unary bitwise complement | |
| *sizeof* | Determine size in bytes | |
| *&* | address | |
| *** | dereference | |
| *new* | Dynamic memory allocation | |
| *new[]* | Dynamic array allocation | |
| *delete* | Dynamic memory deallocation | |
| *delete[]* | Dynamic array deallocation | |
| *(type)* | C-style unary cast | Right to left |
| *.** | Pointer to member via object | Left to right |
| *->** | Pointer to member via pointer | |

# Operator Precedence Table 3

| Operators | Function | Associativity |
|---|---|---|
| * | multiplication | left to right |
| / | division | |
| % | modulus | |
| + | addition | |
| - | subtraction | |
| << | Bitwise left shift | |
| >> | Bitwise right shift | |
| < | Less than | |
| <= | Less than or equal to | |
| > | Greater than | |
| >= | Greater than or equal to | |
| == | Is equal to | |
| =! | Is not equal to | |
| & | Bitwise AND | |
| ^ | Bitwise exclusive OR | |
| \| | Bitwise inclusive OR | |

# Operator Precedence Table 4

| Operators | Function | Associativity |
|---|---|---|
| && | Logical AND | left to right |
| \|\| | Logical OR | |
| ?: | Ternary conditional | Right to left |
| = | Assignment | Right to left |
| += | Addition assignment | |
| -= | Subtraction assignment | |
| *= | Multiplication assignment | |
| /= | Division assignment | |
| %= | Modulus assignment | |
| &= | Bitwise AND assignment | |
| ^= | Bitwise exclusive OR assignment | |
| \|= | Bitwise inclusive OR assignment | |
| <<= | Bitwise left-shift assignment | |
| >>= | Bitwise right-shift assignment | |
| , | sequencing | Left to right |

# Order of Evaluation

- It is important to note that the operator precedence table does not always specify the precise order of evaluation.
- It tells us that **i++*j-–** means **(i++)*(j–-)**
- In this example it does not matter – the outcome will be the same.
- In some cases the order of evaluation is specified – the operands of the logical operators, the ternary conditional and the comma operator are always evaluated from left to right

# Order of Evaluation

- In an expression of the form ***e1*** **?** ***e2*** **:** ***e3*** only one of ***e2*** and ***e3*** will be evaluated;
- If ***e1*** evaluates to true (or a non-zero value) the value of the expression is the value of ***e2*** , otherwise it is the value of ***e3***.
- The logical operators are evaluated lazily: ***e1*** **&&** ***e2*** must be false if ***e1*** is false, so ***e2*** will not be evaluated, and similarly ***e1*****||*****e2*** must be true if ***e1*** is true, so again ***e2*** will not be evaluated.

# The Increment and Decrement Operators

- Both **i++** and **++i** may be used to increment the value held in the variable **i**.
- However the values of the two expressions are different based on their precedence rule
- This difference is significant only if the increment is used as a sub-expression: **a[i++] = 5;** and **a[++i] = 5;** will update different elements of the array.
- The same applies to the **--** operator.
- Note that expressions such as **++i++** and **++i--** are not allowed;

# Classes and Objects – an Example 1

```
1  // Time.h - declaration of class Time.
2  // member functions are defined in Time.cpp
3  #ifndef _TIME_H
4  #define _TIME_H
5
6  class Time
7  {
8      public:
9          Time(); //constructor
10         void setTime(int, int, int);
11         // set hour, minute and second
12         void printUniversal();
13         // print time in universal-time format
14         void printStandard();
15         // print time in standard-time format
16     private:
17         int hour; // 0 - 23 (24-hour clock format)
18         int min; //0-59
19         int sec; //0-59
20 }; //end of class Time
21 #endif
```

# Classes and Objects – an Example 2

```
1  //Time.cpp - member-function definitions for class Time.
2  #include <iostream>
3  #include <iomanip>
4  #include "Time.h"
5
6  using namespace std;
7  // constructor initializes each data member to zero.
8  // ensures all Time objects start in a consistent state.
9  Time::Time()
10 {
11     hour = min = sec = 0;
12 }
13 // set new Time value using universal time;
14 // ensure that the data remains consistent by setting // invalid values to zero
15 void Time::setTime(int h, int m, int s)
16 {
17     hour = (h>=0 && h<24) ? h : 0; // validate hour
18     min = (m>=0 && m<60) ? m : 0;  // validate minute
19     sec = (s>=0 && s<60) ? s : 0;  // validate second
20 }
```

# Classes and Objects – an Example 3

```
1  // Time.cpp (continued)
2  // print time in universal-time format (HH:MM:SS)
3  void Time::printUniversal()
4  {
5      cout << setfill('0') << setw(2) << hour << ":"
6      << setw(2) << min << ":" << setw(2) << sec;
7  }
8  // print time in standard-time format
9  // ([H]H:MM:SS am/pm)
10
11 void Time::printStandard()
12 {
13     cout << (hour==0 || hour==12 ? 12 : hour%12) << ":"
14     << setfill('0') << setw(2) << min << ":"
15     << setw(2) << sec << (hour<12 ? " am" : " pm");
16 }
17
18 /* setw sets the field width of the next item to
19 * be output
20 * setfill sets the fill character to be used for
21 * all subsequent outputs – we need leading zeroes */
```

# Classes and Objects – an Example 4

```
1  // program to test Time class
2  #include <iostream>
3  #include "Time.h"
4
5  using namespace std;
6
7  int main()
8  {
9      Time t; //instantiate object t of class Time
10     // output t's initial value in both formats
11     cout << "The initial universal time is ";
12     t.printUniversal(); // 00:00:00
13     cout << "\nThe initial standard time is ";
14     t.printStandard(); // 12:00:00 am
15     t.setTime(13, 27, 6); //change time
16     // output t's new value in both formats
17     cout << "\n\nUniversal time after setTime is ";
18     t.printUniversal(); // 13:27:06
19     cout << "\nStandard time after setTime is ";
20     t.printStandard(); // 1:27:06 pm
21 }
```

# Invoking Constructors

- When a class object is declared with no initialisation (e.g. **Time t;**)
- If we wish to use a different constructor we must supply the arguments in parentheses after the variable name, e.g. **Time t2(12,15);** .
- To create objects on the heap we would use statements such as

  **Time *tPtr1 = new Time;**

  **Time *tPtr2 = new Time(15,30,40);**
- Note that parentheses are not used if no arguments are to be supplied to the constructor.

# Accessing Class Members

- We have already seen that outside of the member functions of a class its members are accessed using the **.** operator or the **->** operator.
- The left-hand operand of the **.** operator must be a class object or a reference to an object, whereas the left-hand operand of **->** must be a pointer to a class object, e.g.

```
1  Time t;
2  Time tArray[5];
3  Time &tRef = t;
4  Time *tPtr = &t;
5  t.setTime(13, 27, 6);
6  tArray[2].setTime(13, 27, 11);
7  tRef.printStandard();
8  tPtr->setTime(13, 27, 19);
9  (tArray+2)->printUniversal();
```

# Constant Objects and Members Functions

- It is possible to declare constant class objects. e.g.

  **const Time noon(12); // or (12, 0, 0)**

- The only functions that can be applied to such objects are ***constant member functions***.
- Hence a statement such as **noon.printStandard();** would be rejected by the compiler when using the current version of our class.
- To allow such a statement to be able to be used we should have defined **printStandard** as a constant member function by adding the **const** keyword at the end of the declaration:

  **void printStandard() const;**

# Initialising Members of Objects

- Prior to C++11 it was not permissible to initialise a member of a class in its declaration, unless it is a constant static member. Hence

```
1  class X
2  {
3      int y = 3;
4      ......
5  };
```

- would not have been allowed.
- There is an alternative to the initialisation by assignment seen in the constructors for our **Time** class; we can initialise members using an ***initialiser list***:

**Time::Time() : hour(0), sec(0), min(0) { }**

# Initialising Members of Objects

- If one of the members of a class is an object of another class we must invoke one of that class's constructors to initialise it. Consider the following incomplete class.

```
1  class Lecture
2  {
3      private:
4          Time startTime;
5          ......
6      public:
7          Lecture(......, int startHour);
8  };
```

- In the constructor we need to initialise the **startTime** member. We cannot simply write in the body

**startTime(startHour);**

- since the compiler would try to treat this as a function call.

University of Essex

# Initialising Members of Objects

- One option for the initialisation of the **startTime** member would be to allow the no-argument constructor to be invoked implicitly and then use the **setTime** function:

**startTime.setTime(startHour, 0, 0);**

- This is not an ideal approach
- Instead we should perform the initialisation as part of an initialiser list:

```
1  Lecture::Lecture(......, startHour):
2                       startTime(startHour);
3  { ...... }
```

# Initialising Members of Objects

- If a class has as a member an object of another class and has a constructor in which that object is not initialised in an initialiser list, the no-argument constructor for the object is invoked implicitly.
- If the compiler generates a default no-argument constructor for a class, this will initialise any object members using their no- argument constructors.

# Static Members of Classes

- A class may have static members; these belong to the class rather than to individual objects of the class and exist even if no objects of the class exist.
- Consider a class to store information about employees.
- Assume that each employee needs to be given a unique payroll number, with the first being given the number 1, the second 2 and so on.
- Hence we need to keep a count of how many **Employee** objects have been created.

# Static Members of Classes

- The following gives an outline of how we would initialise and use the static member:

```
1  // Employee.h (incomplete)
2  class Employee:
3  {
4      private:
5          int payRollNum;
6          static int count = 0;
7          // = 0 allowed only since C++11 ......
8      public: E
9      mployee(......);
10     ......
11 };
```

```
1  // Employee.cpp (incomplete)
2  // int Employee::count = 0;  need this before C++11
3  Employee::Employee(......)
4  {
5      payRollNum = ++count;
6      ......
7  }
```

# Using this

- If within a member function we need to supply the object to which the function is being applied as an argument to another function we use the keyword **this**, as in Java.
- However, in C++, **this** is a pointer to the object, not a reference.
- If the function being called expects an object or a reference to an object as its argument we have to use ***this**.

# Using this

```
1  #include<iostream>
2  using namespace std;
3
4  /* local variable is same as a member's name */
5  class Test
6  {
7  private:
8     int x;
9  public:
10    void setX (int x)
11    {
12       // The 'this' pointer is used to retrieve the object's x
13       // hidden by the local variable 'x'
14       this->x = x;
15    }
16    void print() { cout << "x = " << this->x << endl;
17    cout << "x = " << (*this).x << endl;
18    }
19 };
```

University of Essex

# CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex