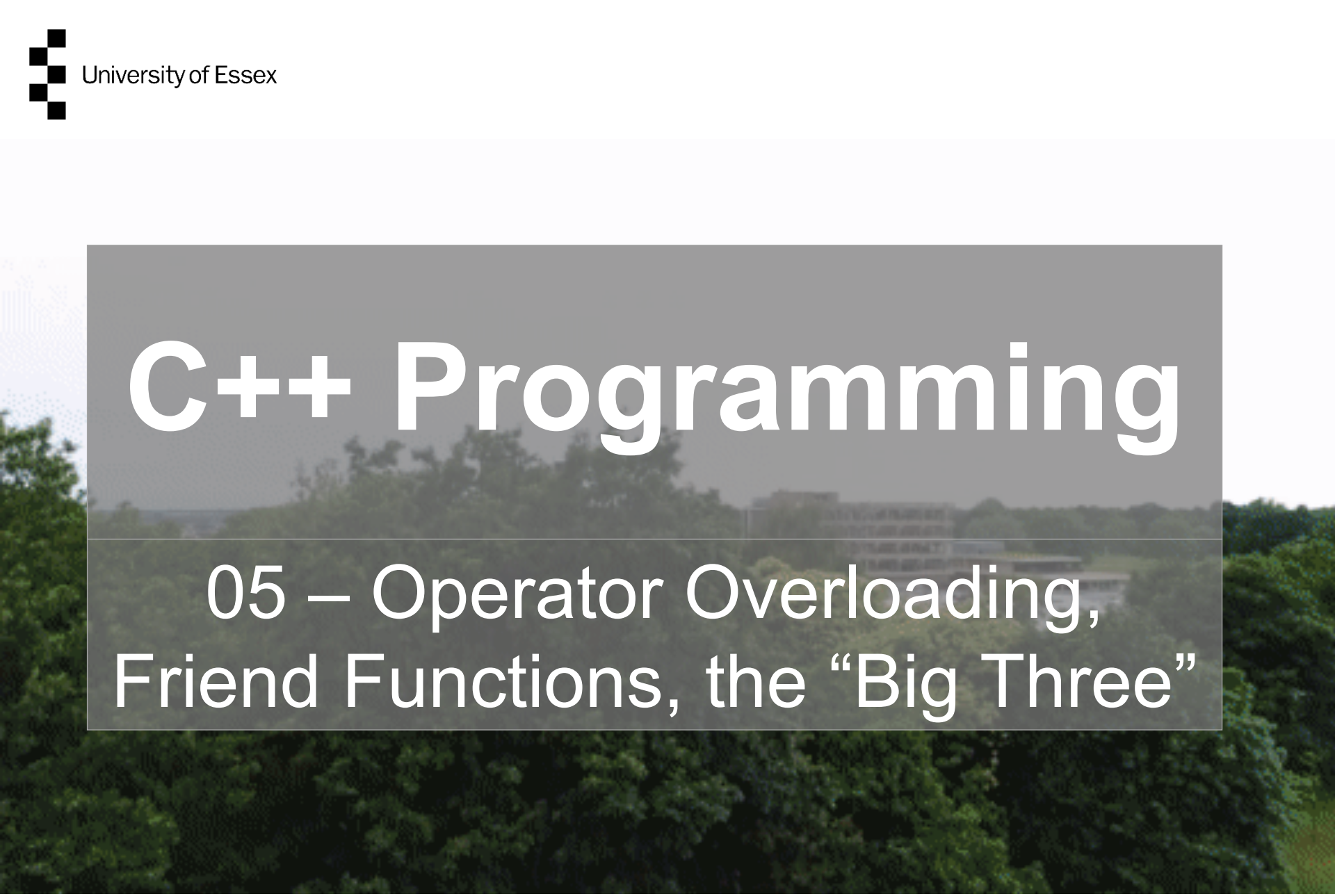
University of Essex



C++ Programming

05 – Operator Overloading, Friend Functions, the “Big Three”

Operator Overloading

- Operator overloading enables the C++ operators to work with objects of user-defined classes.
- Overloading cannot create new operators and cannot change the precedence, associativity or of an operator.
- The four operators **.**, **.***, **::** and **...?...:...** cannot be overloaded.
- The use of **<<** and **>>** for output and input is an example of operator overloading.

Operator Overloading

- When the compiler sees an expression of the form ***e1#e2*** where ***#*** is a binary operator and at least one of ***e1*** and ***e2*** is an expression denoting an object
- It will try to treat the expression as either ***e1*.operator#(*e2*)** or **operator#(*e1*,*e2*)**.
- The value of an overloaded operator is the value returned by the **operator#** function.

Operator Overloading

- If we need to overload any of the operators **()**, **[]** and **->** or any assignment operator for use with a class,
- the function ***must*** be written as a member of the class.
- Note that if we overload an operator such as **+** so that its first argument can be an object of a particular class this does not give a meaning to the corresponding assignment operator (in this case **+=**);

Operator Overloading

- When the compiler sees an expression of the form ***e#*** or ***#e*** where ***#*** where is a unary operator and ***e*** is an expression denoting an object
- It will try to treat the expression as either ***e*.operator#()** or **operator#(*e*)**.
- If we wish to overload either of the **++** or **--** operators for use with objects of a class some method is needed to distinguish between the prefix and postfix versions.
- The compiler will try to treat **++t** as either **t.operator++()** or **operator++(t)**, but will try to treat **t++** as either **t.operator++(0)** or **operator++(t, 0)**.

Operator Overloading - Example

- To give examples of the writing of functions to perform operator overloading we shall provide **+**, **+=** and **++** operators for our time class.
- The value of **t+*n*** where ***n*** is a non-negative integer should be a time ***n*** seconds after the time held in **t**; **t+=*n*** should increment the time stored in **t** by ***n*** seconds and **t++** and **++t** should increment the time stored in **t** by one second.

Operator Overloading - Example

- The **operator+** function must take a parameter of type **unsigned int** and return either an object of type **Time** or a reference to such an object.
- The function must not change the contents of the object to which it is applied so it should be defined as a constant member function.
- The value of the **+=** operator should be a reference to the object that has been assigned to in order to allow it to be used as part of a larger expression such as **(t += 7).printStandard()** or **myt = t += 7**).

Operator Overloading - Example

- Since the value of **++t** is the value of **t** after the increment the prefix version of **operator++** can simply return a reference to the object to which is has been applied, avoiding the need for copying.
- However the value of **t++** is the old value of **t** so we must save a copy of the time in a local variable in the postfix version of the function before incrementing and, since it is unsafe to return a reference to a local variable, we must return a copy of this saved object.
- We should hence add to the public part of the class declaration (in the **.h** file) the lines

```
Time operator+(unsigned int) const;
Time& operator+=(unsigned int);
Time& operator++(); // prefix version
Time operator++(int); // postfix version
```

Operator Overloading - Example

- To avoid duplication of code we should write the code to add to times in one of the functions **operator+** and **operator+=**
- The **+=** operator can be written directly without performing any copying, whereas a copy of the object being returned is made by the **+** operator (since it returns a value rather than a reference).

Operator Overloading - Example

- Here is the complete **operator+=** function.

```
1  Time& Time::operator+=(unsigned int n)
2  {
3      sec += n;
4      if (sec >= 60)
5      {
6          min += sec/60;
7          sec %= 60;
8          if (min >= 60)
9          {
10             hour = (hour + min/60)%24;
11             min %= 60;
12         }
13     }
14     return *this;
15 }
```

- Recall that the value of **this** is a pointer to the object to which the function has been applied; we need to return a reference to the object so we have to use ***this**.

Operator Overloading - Example

- In the **operator+** function we need to make a copy of the object to which the time has been applied, increment this using **+=** and then return it.
- We could make a copy using.

  Time tCopy = *this;

- However this would result in the no-argument constructor being used to initialise an object and the immediate overwriting of the default values using assignment. To be more efficient we should create a copy using

  Time tCopy(*this);

- This invokes a ***copy constructor*** to initialise the new object to be a copy of an existing one.

Operator Overloading - Example

- Here is the code for the **operator+** function and the two versions of **operator++**.

```
1  Time Time::operator+(unsigned int n) const
2  {
3      Time tCopy(*this);
4      tCopy += n;
5      return tCopy;
6  }
7
8  Time& Time::operator++()
9  {
10     *this += 1;
11     return *this;
12 }
13
14 // prefix version
15 Time Time::operator++(int n) // postfix version
16 {
17     Time tCopy(*this);
18     *this += 1;
19     return tCopy;
20 }
```

Friend Functions

- A function defined outside the class but allowed to access the private and protected members of the given class
- Prototype is declared within the class leading with the keyword **friend**

```
3  class A
4  {
5  private:
6    int a, b;
7
8  public:
9    void getdata(int x, int y)
10   {
11     a = x;
12     b = y;
13   }
14   friend void fun1(A);
15 };
16 void fun1(A obj)
17 {
18   cout << "The value of 1st private member is-" << obj.a << endl;
19   cout << "The value of 2nd private member is-" << obj.b << endl;
20 }
```

Friends Functions

- To allow the user to output times using expressions such as **cout<<t** we wish to overload the **<<** operator.
- The compiler will try to treat the above expression as either **cout.operator<<(t)** or **operator<<(cout, t)** so we cannot perform this overloading by writing a member function that is applied to a **Time** object.
- Hence we need to write an **operator<<** function outside the **Time** class.
- In order for this function to be able to access the private data from the class it must be declared as a ***friend function*** inside the class.
- The function will need to take two parameters, references to **ostream** and **Time** objects.
- To allow **cout << t1 << t2** to work as expected the value of **cout << t1** must be **cout**, so the function should return a reference to its stream argument.

Friends Functions

- The **friend** declaration can be made anywhere inside the class declaration, since a friend is not a member of this class and is hence neither private nor public.
- It is however conventional to declare friend functions at either the beginning or end of class declarations:

```
1  class Time
2  {
3      public:
4      ......
5      private:
6      ......
7      friend ostream& operator<<(ostream&,const Time&);
8  };
```

Friends Function

- The **operator<<** function is simply written as a global function in the **.cpp** file.
- The body of the function should be very similar to that of one of the **print** functions written in part 3; we choose to use the universal version.

```
1  ostream &operator<<(ostream &o, const Time &t)
2  {
3      o << setfill('0') << setw(2) << t.hour << ":"
4      << setw(2) << t.min << ":" << setw(2) << t.sec; return o;
5  }
6
```

- We might wish to overload the **>>** operator to allow for input of times but to provide a robust version of this

Constant References

- Assume that the following three statements appear in separate parts of a program.

```
const int months = 12;
int &a = months;
a = 13;
```

- This cannot be allowed since it enables the value of a constant to be changed.
- Hence only a constant reference may refer to a constant data item so we need

```
const int &a = months;
```

Constant References

- A constant reference may refer to a non-constant data item so the following code is allowed.

```
int x = 99;

const int &a = x;
```

- If this code does appear in a program we are not allowed to change the contents of the data item by using the reference, so although **x = 101;** would be allowed **a = 101;** would not.

Constant References

- If a function returns a non-constant reference a call to that function may be used as the left-hand operand of an assignment, e.g.

  f(a) = 99;

- To prevent a statement such as

  s.regNo() = 123;

- being used to change the value of an attribute of a constant object, a constant member function is not allowed to return a non-constant reference to the object to which it is applied or any of the members of that object.

The “Big Three”

- All classes must have a copy constructor, an assignment operator and a ***destructor***.
- These three functions are often referred to as the “big three”.
- If the programmer fails to provide any of these functions for a class the compiler will generate a default version.

The “Big Three”

- The assignment operator for a class **X** is a public member function declared as **X& operator=(const X&)**.
- It is called implicitly whenever a programmer uses an assignment of the form **x = *e***, where **x** refers to an object of the class and ***e*** is an expression whose value is an object of the class.
- Note that is permissible to write other **operator=** functions with different argument types.

The “Big Three”

- A destructor for a class **X** must be a public member function called **~X**.
- It must have no arguments and has no return type.

The “Big Three”

- The copy constructor for the class takes as an argument another object of the class and is invoked implicitly in all circumstances where a copy of a class object is needed other than by assignment.
- The argument to a copy constructor must be a reference; otherwise we would have to use the copy constructor to create an object to be passed as an argument to the copy constructor.
- It must not change the contents of the object that is being copied so the copy constructor for a class **X** should be declared as **X(const X&)**.

The “Big Three”

- The default copy constructor generated by the compiler will use the copy constructors for any members of the class which are objects of other classes and initialise all other members using simple copying.
- The default assignment operator will just perform assignments to all the members.
- The default destructor will invoke the destructors for any members of the class which are objects of other classes and do nothing else.

The “Big Three”

- The most usual circumstances in which non-default versions are required are when one of the members of the class is a pointer.
- The copy of the pointer will point to the same data item as the original pointer.
- This is sometimes referred to as ***shallow copying***; in many cases ***deep copying*** is required, i.e. the pointer in the copy should point to a copy of the data item to which the pointer in the original object pointed.

Example – an Array Class

- The standard library provides a class for arrays with range- checking.
- This is called **vector** and allows for arrays of objects of any type.
- To illustrate the use of the “big three” we shall show how to write a similar, but simpler class, which just supports arrays of **int**.
- When we make a copy of an **Array** object we need to create a copy of the dynamic array so the default copy constructor and assignment operator are not appropriate.

Example – an Array Class

- Note that to allow the use of **a[i]** in different contexts we need to provide two versions of the **operator[]** function.
- If **a** is a constant array we need to allow the use of expressions such as **x = a[i]**.
- To allow the operator to be used in an assignment expression such as **a[i] = 3** when **a** is not a constant array we also need a version that returns a non-constant reference.

Example – an Array Class

```
1  // // Array.h
2  #ifndef _ARRAY_H_
3  #define _ARRAY_H_
4
5  #include <iostream>
6
7  using namespace std;
8
9  class Array
10 {
11  public:
12      Array(int = 10); // size; default argument
13      Array(const Array&); // copy constructor
14      ~Array(); // destructor
15      const Array& operator=(const Array&);
16      int getSize() const;
17      int& operator[](int);
18      int operator[](int) const;
19  private:
20      int size;
21      int *ptr; // pointer to first element
22  friend ostream &operator<<(ostream&,const Array&);
23  friend istream &operator>>(istream&, Array&);
24          // will not be robust
25 }; #endif
```

Example - an Array Class

- The assignment operator should return a reference to the object to which it is applied.
- We chose to return a constant reference in order to prevent attempts to use expressions such as **(a1 = a2) = a3**.
- We also provided **operator==** and **operator!=** functions which checked whether two arrays had the same contents.

Example – an Array Class

```cpp
// Array.cpp
#include <iostream>
#include <iomanip>
#include "Array.h"

using namespace std;

Array::Array(int arrSize)
{
    size = arrSize>0 ? arrSize : 10;
    ptr = new int[size];
    for (int i = 0; i<size; i++)
        ptr[i] = 0;
}

Array::Array(const Array &a): size(a.size)
{
    ptr = new int[size];
    for (int i = 0; i<size; i++)
        ptr[i] = a.ptr[i];
}
```

```cpp
Array::~Array()
{
 delete [] ptr;
}

const Array& Array::operator=(const Array &a)
{
 if (&a != this)
 {
     if (size != a.size)
     {
         delete [] ptr;
         size = a.size;
         ptr = new int[size];
     }
     for (int i = 0; i<size; i++)
         ptr[i] = a.ptr[i];
 }
 return *this;
}
```

University of Essex

Example – an Array Class

- If the argument supplied to the assignment operator is a reference to the same array as the object to which the function is applied (e.g. **a = a**) no copying is necessary.
- If the argument has the same size as the object to which the function is applied there is no need to allocate a new dynamic array;

Example – an Array Class

```
1  int& Array::operator[](int subscript)
2  {
3      assert(0 << subscript && subscript< size);
4      return ptr[subscript];
5  }
6
7  int Array::operator[](int subscript) const
8  {
9      assert(0 << subscript && subscript< size);
10     return ptr[subscript];
11 }
12 /* the assert function takes a boolean argument and
13 * throws an exception if value of the argument
14 * is not true; otherwise it does nothing
15 */
```

```
16 istream& operator>>(istream &in, Array &a)
17 {
18     for (int i = 0; i<a.size; i++)
19         in >> a.ptr[i];
20     return in;
21 }
22
23 ostream& <<(ostream &out, const Array &a)
24 // displays 4 items per line in fixed-width fields
25 {
26     int i;
27     for (i = 0; i<a.size; i++)
28     {
29         out << setw(12) << a.ptr[i];
30         if (i%4 == 3) out << endl;
31     }
32     if (i%4 != 0)
33         out << endl;
34     return out;
35 }
```

Type Conversion

- If a class has a member function **operator *T*()** where ***T*** is the name of a type, this function may be used to convert an object of the class to an object of type ***T***.
- Such a function must return a result of type ***T***.
- We can use this, for example, to add to our **Time** class something similar to Java's **toString** method.
- We would add to the public part of the class declaration

operator string();

or

operator char*();

- The type is not specified in the declaration since the compiler can infer that **operator string()** must return a string.

Type Conversion

- To invoke the functions on our previous slide to convert a time **t** into a string we would simply use **char*(t)** or **string(t)**.
- Type conversion operator functions can be invoked implicitly – if we provided an expression denoting an object of type **Time** in some context where an object of type **string** is required the function **operator string** would be applied to the object.
- Hence having written this function permissible to write code such as **string s = t;** or supply a time as an argument to a function that has a **string** parameter.
- Only one conversion will be performed implicitly; if there are conversion operators to convert from **T1** to **T2** and **T2** to **T3** we cannot use an object of type **T1** where **T3** is expected.

Type Conversion Constructors

- A constructor with a single argument of a type other than the class of which it is a member can be used as a implicit type conversion constructor.
- Hence, since the **Array** class has a constructor with an **int** parameter we could use an **int** in a program in any context in which an array is expected.
- To prevent implicit conversions using a constructor we should declare that constructor as explicit in the class declaration:

explicit Array(int=10);

University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex