CE221

Programming in C++
Part 4
Operator Overloading, Friend Functions, the “Big Three”
Operator Overloading 1

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 arity (i.e. how many operands it has) of an operator.

The four operators ., .*, :: and ...?...:... cannot be overloaded.

The use of << and >> for output and input is an example of operator overloading; when used with primitive types these operators denote bit-shifting but they have been overloaded to mean input or output when the first operand is a stream object.
Operator Overloading 2

When the compiler sees an expression of the form $e_1 \# e_2$ where $\#$ is a binary operator and at least one of $e_1$ and $e_2$ is an expression denoting an object, it will try to treat the expression as either $e_1.\text{operator}\#(e_2)$ or $\text{operator}\#(e_1, e_2)$.

Hence, if we wished to give a meaning to $t+7$ where $t$ is an object of the Time class developed in part 3 we would have to write a function called $\text{operator+}$ either as a member of the class (with a single integer argument) or as a two-argument function with global scope.

The value of an overloaded operator is the value returned by the $\text{operator}\#$ function.
Operator Overloading 3

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 +=); if we wished to give a meaning to t += 7 we would have to write a function operator+= as a member of the Time class.

When the compiler sees an expression of the form e# or #e where # where is a binary operator and e is an expression denoting an object, it will try to treat the expression as either e.operator#() or operator#(e).
Operator Overloading 4

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)`. Hence to overload the postfix version we need to write a function with an extra `int` argument. This argument is just a dummy to allow the compiler to make the distinction; we should not expect to access it inside the body of the function.
Operator Overloading – an Example 1

To give examples of the writing of functions to perform operator overloading we shall provide +, += and ++ operators for our time class. (We could also provide -, -= and -- operators; the code would be similar.)

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. We want these to behave in the same way as the ++ operators for integers so the value of \( t++ \) should be the old value of \( t \) and the value of \( ++t \) should be the new value of \( t \).
Operator Overloading – an Example 2

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. Since it is unsafe to return a reference to a local variable we return an object rather than a reference. 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`). Hence the `operator+=` function should return a reference to an object of type `Time`. 
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

```cpp
Time operator+(unsigned int) const;
Time& operator+=(unsigned int);
Time& operator++();   // prefix version
Time operator++(int); // postfix version
```
Operator Overloading – an Example 4

To avoid duplication of code we should write the code to add to times (which may involve updating hours, minutes and seconds) in one of the functions `operator+` and `operator+=` and then make use of the newly-defined operator to implement the other three functions.

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). Implementing the `+=` operator using `+` would hence introduce unnecessary copying so we choose to implement `+=` directly and use it in the implementation of `+` and the two versions of `++`. 
Here is the complete `operator+=` function.

```cpp
Time& Time::operator+=(unsigned int n)
{
    sec += n;
    if (sec >= 60)
    {
        min += sec/60;
        sec %= 60;
        if (min >= 60)
        {
            hour = (hour + min/60)%24;
            min %= 60;
        }
    }
    return *this;
}
```

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 – an Example 6

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:

```cpp
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:

```cpp
Time tCopy(*this);
```

This invokes a `copy constructor` to initialise the new object to be a copy of an existing one.
Operator Overloading – an Example 7

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

```cpp
Time Time::operator+(unsigned int n) const
{
    Time tCopy(*this);
    tCopy += n;
    return tCopy;
}

Time& Time::operator++() // prefix version
{
    *this += 1;
    return *this;
}

Time Time::operator++(int n) // postfix version
{
    Time tCopy(*this);
    *this += 1;
    return tCopy;
}
```
Friend Functions 1

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.
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:

```cpp
class Time
{
    public:
        // ...
    private:
        // ...
friend ostream& operator<<(ostream&, const Time&);
};
```
Friend Functions 3

The **operator<<** function is simply written as a global function in the `.cpp` file; it is not a member of the class so we must not use `Time::operator<<`. 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.

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

We might wish to overload the `>>` operator to allow for input of times but to provide a robust version of this we would need to input the time as a string and extract the relevant parts – we do not yet know how to do this.
Assume that the following three statements appear in separate parts of a program.

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

This cannot be allowed since it enables the value of a constant to be changed. The fact that the three statements appear in different parts of a program (and possibly in different files) means that the compiler cannot check at the point of the assignment whether the data item to which `a` refers is a constant so the second line has to be forbidden. Hence only a constant reference may refer to a constant data item so we need

```c
const int &a = months;
```
Constant References 2

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

```c
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.

[ It is necessary to allow constant references to non-constant data items so that when a function has a constant reference as a parameter we can supply a reference to a non-constant data item as an argument in a call to that function. ]
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. (It may of course return a non-constant reference to another data item.)
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. (Note that although the compiler will not generate a default no-argument constructor if the programmer has provided any constructors it will always provide a default copy constructor if the programmer has not provided one.)
The “Big Three” 2

The assignment operator for a class \texttt{x} is a public member function declared as \texttt{X\& operator=(const X\&)}. It is called implicitly whenever a programmer uses an assignment of the form \texttt{x = e}, where \texttt{x} refers to an object of the class and \texttt{e} is an expression whose value is an object of the class.

Note that is permissible to write other \texttt{operator=} functions with different argument types. If one or more such functions are written but the assignment operator is not, the compiler will still generate a default assignment operator.
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 destructor of a class is applied to any object immediately before its lifetime expires, i.e. at the end of a block or function for a local variable, at the end of the program for a global variable and when the memory space for a dynamically-created object is returned to the heap using the `delete` (or `delete []`) operator.
The “Big Three” 4

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. These include the passing of non-reference arguments to functions and the returning of non-reference results from functions as well as any initialisation of an object using an argument of the same type (as used on slide 12).

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. Hence the copy constructor for a class \texttt{x} must be declared as \texttt{X(const \texttt{x}&)}.
The “Big Three” 5

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.

In many cases, including our Time class, the default behaviour is what is wanted so the programmer does not need to write his own versions.
It is almost always the case that if it is necessary to provide a non-default version of one of the “big three” for a class it will be necessary to provide non-default versions of the others as well.

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 1

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`. This is based on an example from Deitel & Deitel.

The data members of the class will be a pointer to the first element of a dynamically-allocated array and the length of the array. 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.
// Array.h

#ifndef __ARRAY_H__
#define __ARRAY_H__

#include <iostream>
using namespace std;

class Array
{
   public:
      Array(int = 10);          // size; default argument
      Array(const Array&);     // copy constructor
      ~Array();                 // destructor
      const Array& operator=(const Array&);    // assignment operator

   // continued on next slide
Example – an Array Class 3

// Array.h continued
// public members of class Array continued

    int getSize() const;
    int& operator[] (int);
    int operator[](int) const;

private:
    int size;
    int *ptr; // pointer to first element

friend ostream &operator<< (ostream&,
                            const Array&);
friend istream &operator>>(istream&, Array&);

};
#endif
Example – an Array Class 4

Note that to allow the use of \( a[i] \) in different contexts we need to provide two versions of the \texttt{operator[]} function.

If \( a \) is a constant array we need to allow \( x = a[i] \) (as long as \( x \) is not a reference variable). The only functions that can be applied to constant objects are constant member functions so we need to supply a constant member function. This cannot return a non-constant reference to a member of the object to which it is applied so it must return a constant reference or a copy of the member; we choose to return a copy.

To allow the operator to be used in an assignment expression such as \( a[i] = 3 \) when \( a \) is not a constant array we need a version that returns a non-constant reference. Hence we need two distinct versions.
The assignment operator has to return a reference to the object to which it is applied. Deitel & Deitel chose to return a constant reference in order to prevent attempts to use expressions such as \((a_1 = a_2) = a_3\).

It would be possible to add more member functions; Deitel & Deitel also provided `operator==` and `operator!=` functions which checked whether two arrays had the same contents.
/ 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];
}
// Array.cpp continued

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;
}
Example – an Array Class 8

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; we can overwrite the contents of the existing one.

We could have chosen to allocate a new dynamic array only if the size of the argument was larger than the existing dynamic array; using this approach we would need to store both the current array size and the dynamic array size as separate members of the class.
// Array.cpp continued

int& Array::operator[](int subscript)
{
    assert(0 <= subscript && subscript < size);
    return ptr[subscript];
}

int Array::operator[](int subscript) const
{
    assert(0 <= subscript && subscript < size);
    return ptr[subscript];
}

/* the assert function takes a boolean argument and
 * throws an exception if value of the argument
 * is not true; otherwise it does nothing
 */
Example – an Array Class 10

// Array.cpp continued

istream& operator>>(istream &in, Array &a)
{ for (int i = 0; i<a.size; i++)
    in >> a.ptr[i];
    return in;
}

ostream& <<(ostream &out, const Array &a)
// displays 4 items per line in fixed-width fields
{ int i;
    for (i = 0; i<a.size; i++)
    { out << setw(12) << a.ptr[i];
      if (i%4 == 3) out << endl;
    }
    if (i%4 != 0) out << endl;
    return out;
}
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:

```cpp
    string operator string();
```

or

```cpp
    char* operator char*();
```
Type Conversion 2

To invoke the functions on our previous slide to convert a time \( t \) into a string we would simply use \( \text{char}^{*}(t) \) or \( \text{string}(t) \).

Type conversion operator functions can be invoked implicitly – if we provided an expression denoting an object of type \( \text{Time} \) in some context where an object of type \( \text{string} \) is required the function \( \text{operator string} \) would be applied to the object. Hence having written this function permissible to write code such as \( \text{string} \ s = t; \) or supply a time as an argument to a function that has a \( \text{string} \) parameter.

Only one conversion will be performed implicitly; if there are conversion operators to convert from \( T_1 \) to \( T_2 \) and \( T_2 \) to \( T_3 \) we cannot use an object of type \( T_1 \) where \( T_3 \) 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 an 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. We might consider this as undesirable. If a programmer uses an integer when an array is expected he has probably made a mistake and we would want a syntax error to be reported. To prevent implicit conversions using a constructor we should declare that constructor as explicit in the class declaration:

```cpp
explicit Array(int=10);
```