# Introduction to C++ (part 2)

(00:00)

Welcome back to our coding session where we are writing a class representing data about computer science lecturers in C++. In the previous video, we wrote our constructors, setters and getters, and made some comparisons between C++ and Java.

Now we'll move on to complete our implementation.

## Friends and overloaded operators

One of the things we'd like to do is make it easy to print information about a lecturer in some standard format. Say we wanted to print a table of lecturers, giving their name, the courses they teach, and their office address. That is, it would be super convenient if we could write something like this

```
for (Lecturer& lec : lecturers) {
    std::cout << lec;
}
```

to print a list of lecturers, their courses, and their office addresses in a common format. As it is, we cannot do that, because inserting ( `<<` ) a lecturer object into the `cout` stream will result in an error. There are a few ways we could handle this, but one of the most elegant is to override the insertion operator ( `<<` ) when used with an object of class Lecturer.

First, what does it mean to "override" an operator? It means we're going to take an existing operator ( `<<` ) and give it a new meaning in the context of our class. Now for this to make sense, the overloaded operator should behave in a similar way and it should have the same number of operands. In the case of the insertion operator, it takes the operand to the right and inserts it into the output stream (on the left). Our overloaded operator should do the same thing, but for our class, `Lecturer`. That is, it should insert a Lecturer object into the `cout` stream.

Now, how do we accomplish this? We want the `std::ostream` insertion operator `<<` to be made "aware" of the Lecturer class. In order to assemble all the strings it needs for the output, it will need access to the fields of the Lecturer class. So we begin our declaration with the friend keyword.

A friend function is not strictly a member function, but is one which has access to private and protected fields within the class. (Accordingly, it is part of the class interface.)

So our declaration will look like this:

```
friend std::ostream& operator << (std::ostream& outs, const Lecturer &lec) {...}
```

Here we're stating that we have a friend, overriding the output stream insertion operator, and that it will take a Lecturer object and not modify it (hence the `const` keyword).

Now let's flesh this out. When given a `Lecturer` object, we wish to print five columns: the lecturer's name, the three courses they teach, and their office address. We have a reference to the ostream object `outs`. To this we wish to pass our formatted data. This is as follows:

- a column of width 30 to hold the lecturer's name
- three columns to hold the course numbers, and
- a column of width 20 to hold the office address.

Here's how we implement this.

```
friend std::ostream& operator << (std::ostream& outs, const Lecturer& lec) {
    outs << std::setw(30) << lec.getName()
         << std::setw(5) << lec.getCourse1()
         << std::setw(5) << lec.getCourse2()
         << std::setw(8) << lec.getCourse3()
         << std::setw(20) << lec.getOffice();
    return outs;
}
```

Notice that the friend — the overloaded operator — has access to `Lecturer` class methods. When we've supplied all the formatted output, we return the reference to the output stream. That's it. Now we have overridden the insertion operator so it "knows" what to do when presented with an object of class `Lecturer`. That's kinda cool, huh?

(08:02)

There's an opportunity to use operator overloading in another case as well. Let's say we wanted to compare lecturers in some way, like we might compare two numbers. In the case of numbers we could use the less than operator, `<`, to compare two numbers. This would return a boolean: true, if the left operand were less than the right operand, and false otherwise. Let's do something similar with lecturers.

Now, there are lots of ways we could perform comparisons between lecturers, but let's do it on the basis of the length of their names.

Again, this will be a friend, it will return a boolean, it will override the less than operator, `<`, and it will take two operands: a reference to the lecturer on the left-hand side and a reference to the lecturer on the right-hand side. So our declaration will look like this:

```
friend bool operator < (const Lecturer& lhs, const Lecturer& rhs) {
    ...
}
```

Since we want to compare the lengths of the two names, we can use the string's `length()` method. So our code becomes,

```
friend bool operator < (const Lecturer& lhs, const Lecturer& rhs) {
    return lhs.getName().length() < rhs.getName().length();
}
```

That's it. Now through operator overloading we've made the less than operator aware of `Lecturer` objects.

## Test function

(11:00)

Let's write a test function to make sure things behave as we expect.

Ideally we'd write a separate program file to test our class, but here our class is relatively small, and we don't want to get into compiling multiple targets right now, so we'll just write an auxiliary function here. Our first test will make sure that the default constructor and the overloaded insert operator work as intended. We create a new lecturer object then we check to see if the result is what we'd expect.

```
void testLecturerClass() {
    // Test default constructor
    Lecturer lec;
    // Tests overloaded << operator
    std::cout << lec;
    assert(lec.getName() == "Jane Doe");
    assert(lec.getCourse1() == 0);
    assert(lec.getCourse2() == 0);
    assert(lec.getCourse3() == 0);
    assert(lec.getOffice() == "Innovation E200");
...
}
```

Now notice that in addition to printing test output, we have used assertions. We "assert" something that we believe to be true if our code is working correctly, that is: the name should equal "Jane Doe", the course numbers should all be zero, and the office should equal "Innovation E200". If any of these assertions fails, our program will raise an exception. Assertions are very useful in testing our code.

The usage of assertions is to write "assert," and then in parentheses we give some statement we expect to evaluate to true.

Next, we'll test the other constructor, and the overridden insert operator (again).

```
void testLecturerClass() {
...
    // Test non-default constructor
    Lecturer lec2("Lisa Dion", "Innovation E314",
            120, 124, 292);
    std::cout << lec2;
    assert(lec2.getName() == "Lisa Dion");
    assert(lec2.getCourse1() == 120);
    assert(lec2.getCourse2() == 124);
    assert(lec2.getCourse3() == 292);
    assert(lec2.getOffice() == "Innovation E314");
...
}
```

Again, we've used assertions to verify that our expectations are met.

Now we test a getter / setter pair.

```
void testLecturerClass() {
...
    // Test getters and setters
    lec.setName("Spongebob");
    std::cout << lec.getName() << std::endl;
    assert(lec.getName() == "Spongebob");
...
}
```

And finally, we test the overloaded `<` operator.

```
void testLecturerClass() {
...
    // Test overloaded < operator
    std::cout << std::boolalpha << (lec < lec2) << std::endl;
    assert(lec < lec2 == false);
    lec.setName("Eugene");
    std::cout << std::boolalpha << (lec < lec2) << std::endl;
    assert(lec < lec2 == true);
...
}
```

Here, we provide two test cases — one where we expect the answer to be false, one where we expect the answer to be true.

What is `std::boolalpha`, you ask? It's just an output manipulator to allow for `true` and `false` values to be displayed as "true" and "false" rather than as "1" and "0". It's not essential, but it makes our output a little prettier.

With these tests in place, we have confidence that our `Lecturer` class and its methods are working as they should.

There's an old adage: "Untested code is broken code" so be liberal with your testing and use assertions!

## Importing data from CSV

(20:47)

Finally, we'll write a utility function to import lecturer data from a CSV file. This is where we'll use the `vector` and `fstream` libraries we included at the beginning.

To begin, let's look at what a data file looks like, so we understand what we're working with.

```
Lecturer Name,Course 1,Course 2,Course 3,Office
Radhakrishna Dasari,20,21,,Innovation E305
Lisa Dion,120,124,292,Innovation E314
James (Jim) Eddy,21,121,166,Innovation E309
Robert (Bob) Erickson,8,148,,Innovation E304
Jason Hibbeler,201,205,275,Innovation E315
Jackie Horton,21,110,,Innovation E318
```

Notice that there's a heading row at the top, which includes field names. Then each line consists of a lecturer's name, the courses taught by that lecturer, and the lecturer's office address. These values are separated by commas. Notice that not all lecturers teach three different courses, so we have some null values — that is, an empty string between commas.

So this is what we want to parse and process.

Let's define a function to do this. Our function will take two parameters: a string indicating the filename, and a `vector` object, to hold our data.

We begin by instantiating a new `ifstream` object, which we'll use to operate on our input file.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
    std::ifstream fIn;
...
}
```

Then we open the file, and read the first line of the file into a variable called `header`.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
...
    fIn.open("../" + filename);
    if (fIn) {
        std::string header = "";
        getline(fIn, header);
    }
...
}
```

Notice that we prefix the filename with "../". Here we're assuming that the executable is in a separate build directory, and that the CSV file is one level up from this. If we build using CLion (as we're going to do shortly), the default folder for the executable will be called `cmake-build-debug`.

Next we create a `while` loop to consume our CSV file one line at a time, and we make sure we close our file when done.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
...
    while (fIn && fIn.peek() != EOF) {
        ...
    }

    fIn.close();
...
}
```

Now what goes in the while loop? At each iteration, we'll want to read a line, breaking it up by commas. Then we'll want to extract values from text and store them in variables. When we have all variables for a line of data populated we'll add a new `Lecturer` object to our vector, using the `push_back` method. Let's do this a step at a time.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
...
    while (fIn && fIn.peek() != EOF) {
        std::string name = "", office = "";
        int course1 = 0, course2 = 0, course3 = 0;
        char comma = ',';
        ...
    }
...
}
```

Let's declare some variables we'll use inside this loop. We'll want placeholders for our data: `name` , `office` — these are strings, and notice we can declare them in the same line. Ditto for `course1` , `course2` and `course3` , except that these are `int` s. For convenience we'll define the character variable comma to be a single comma. This will serve as our delimiter. Notice that strings appear in double quotes and characters in single quotes.

Now we'll use the getline function to read from our file one line at a time.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
...
    while (fIn && fIn.peek() != EOF) {
        std::string name = "", office = "";
        int course1 = 0, course2 = 0, course3 = 0;
        char comma = ',';

        getline(fIn, name, comma);
        ...
    }
...
}
```

`getline` reads a line up to the delimiter (if supplied) or a newline character or the end of the file, whichever comes first. So this will store a line, up to but not including the first comma, into the name variable.

Let's get some more data.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
...
    while (fIn && fIn.peek() != EOF) {
        ...
        getline(fIn, name, comma);

        fIn >> course1;
        fIn >> comma;

        fIn >> course2;
        fIn >> comma;

        fIn >> course3;
        if (!fIn) {
            // no integer found for course3
            course3 = 0;
            fIn.clear();
        }
        fIn >> comma;
    }
...
}
```

Here we get at the data a different way, using the extraction operator, `>>`. As the insert operator, `<<`, inserts data into a stream, the extraction operator extracts data. Using this, we fetch data for courses taught. But notice that some lecturers only teach two different courses, so we need some special handling. Here we check to see if we got any data when extracting from `fIn`. If we did not, then `fIn`, will have error flags set indicating it is not in a "good" state, and when we check, it evaluates to `false`. In this event, we supply a default value of 0, and then call the `clear()` method on `fIn`. This resets internal error state flags for the `fIn` object, so that we can continue.

Next we expect to find a comma. OK.

Then finally we use `getline` again, to fetch data from the last field in our CSV file. Notice that here we don't specify a delimiter so getline reads until it hits a newline character.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
...
    while (fIn && fIn.peek() != EOF) {
        ...
        getline(fIn, office);
        ...
    }
...
}
```

So now we have all our variables populated with data read from file, and we can create a new lecturer object and push it into our vector.

Here we use the complete constructor for `Lecturer`, and then the `push_back` method of our vector. This appends a new element — a `Lecturer` object to the end of the vector.

```
void readLecturersFromFile(std::string filename,
                           std::vector<Lecturer>& lecturers) {
...
    while (fIn && fIn.peek() != EOF) {
        ...
        Lecturer lec(name, office, course1, course2, course3);
        lecturers.push_back(lec);
    }
...
}
```

## A program using the `Lecturer` class

(31:43)

Now we're done with our header file. Now let's write a little program that does the following.

1. Calls our `testLecturerClass()` function.
2. Displays a column heading for tabular display.
3. Reads from the CSV data file and then prints a table of lecturer data.

If all this works OK, then we've done our job.

We'll begin this last part by creating a file called `main.cpp` .

(Create `main.cpp` )

We'll include our lecturer header, and headers for I/O manipulation and streaming.

```
#include <iomanip>
#include <iostream>
#include "Lecturer.h"
```

Notice that when we include our own header file, the name is in quotes and not in angle brackets.

Now we'll write our `main()` function. Every C++ program should have one, and note that this is global, and not part of any class.

```
int main() {
    ...
}
```

Next we'll configure our output fills to the left. This will be used to format our output.

```
int main() {
    std::cout << std::left;
    ...
}
```

Then we'll call our test function.

```
int main() {
    std::cout << std::left;
    testLecturerClass();
    ...
}
```

When the program is run this function will be called and will generate its own output. So all there is left to do is print a table heading and generate the table from data extracted from the CSV file.

```
int main() {
    ...
    std::cout << std::setfill('_');
    std::cout << std::setw(30) << "Lecturer" << std::setw(18) << "Classes"
        << std::setw(20) << "Office" << std::endl;
    ...
}
```

Notice how we've set the fill character to an underscore, and set the widths of our columns to mirror the widths we used in our overridden output function.

Now we switch our fill character — since we don't want all those underscores in the body of the table.

```
int main() {
    ...
    std::cout << std::setfill(' ');
    ...
}
```

Then we create a vector to hold our lecturer data. We do this with the standard vector library. We'll be passing this object to our `readLecturersFromFile()` function in a moment.

```
int main() {
    ...
    std::vector<Lecturer> lecturers;
    ...
}
```

Notice that this creates an vector for holding `Lecturer` objects. We'll call it "lecturers" (makes sense, right?).

Now we'll call our `readLecturersFromFile()` function.

```
int main() {
    ...
    std::vector<Lecturer> lecturers;
    readLecturersFromFile("cs_lecturers.csv", lecturers);
    ...
}
```

This will populate our `lecturers` vector in place. Once it's populated, we'll want to iterate through the elements in the vector and then print each one in our table.

```
int main() {
    ...
    for (Lecturer& lec : lecturers) {
        std::cout << lec << std::endl;
    }
    ...
}
```

Here's a `for` loop that understands how to iterate a vector, and for each iteration it provides a `Lecturer` object called `lec`. At each iteration, we insert this into the output stream using our overridden operator — see, how convenient that turned out to be?

Finally, we'll print the size of the vector

```
int main() {
    ...
    std::cout << "Size of vector: " << lecturers.size() << std::endl;
    ...
}
```

Oh, and let's not forget to return zero!

```
int main() {
    ...
    return 0;
}
```

Now we're ready to build and check for warnings or errors.

Let's check our `CMakeLists.txt` file to make sure it's okay.

```
make_minimum_required(VERSION 3.12)
project(Lecturer)

set(CMAKE_CXX_STANDARD 14)

add_executable(lecturer main.cpp Lecturer.h)
```

That looks good. We have a project "Lecturer" and we're going to build an executable called `lecturer` from the source file `main.cpp` and `lecturer.h`.

From the CLion menu we run our build. That looks OK.

Now we'll run our program and inspect the output.

Hey, that looks great. The output of our test function looks as we'd expect:

```
Jane Doe                      0    0    0       Innovation E200
Lisa Dion                     120  124  292     Innovation E314
Spongebob
false
true
```

and our output table looks good too.

```
Lecturer______________________Classes___________Office______________
Radhakrishna Dasari           20   21   0       Innovation E305
Lisa Dion                     120  124  292     Innovation E314
James (Jim) Eddy              21   121  166     Innovation E309
Robert (Bob) Ericson          8    148  0       Innovation E304
Jason Hibbeler                201  205  275     Innovation E315
Jackie Horton                 21   110  0       Innovation E318
Size of vector: 6
```

## Conclusion

The code for this exercise has been posted with additional comments to Blackboard. I encourage you to read through it to remind yourself of the steps we took here. If you find some code you don't understand, read the additional comments, or just come back to the appropriate point in this video.

I know we've covered a lot of material here, but I hope you won't find the transition to C++ daunting. In fact, as we saw earlier, there are a lot of similarities to Java.

If you have additional questions, please don't hesitate to contact your instructor or TA. Thanks for your time, and GOOD LUCK!