Introduction to C++ (part 1)

(00:00)

Introduction to C++

Our objective here is to demonstrate how to define and implement a class in C++.

Ease transition with comparison between C++ and Java

We assume you've taken CS 110 or the equivalent and therefore have some familiarity with Java. So we'll start with a quick look at an implementation of a simple node class for linked lists in Java, then we'll code up a class in C++ to represent data about computer science lecturers, and we'll perform a little comparison between the two.

First, let's look at `Node.java`. This is a class that's intended to represent nodes in a singly-linked list.

(Open `Node.java` in IDE)

If you haven't seen linked lists before, don't fret. A linked list is merely a collection of nodes in which each node contains some information and a pointer or reference to the next node in the list. You may think of this data structure as beads on a string, with each bead pointing to the next bead on the string.

So each node will require a minimum of two fields: one to hold some data and one to hold some kind of reference to the next node in the list. Let's look at the code.

Not surprisingly, we begin with a class declaration with the class keyword and an identifier "Node". Then we declare two private fields: `item`, which is a generic object, and `next` which is a `Node` object.

```
public class Node {
    private Object item;
    private Node next;
...
}
```

`item` will hold the "data", and `node` will hold a reference to the next node.

Then we declare a number of public methods.

First there's a constructor for the `Node` object. We pass to the constructor an item which is assigned to the `item` field. The `next` field is given the value `null`.

```
public Node(Object newItem) {
    item = newItem;
    next = null;
}
```

There's another constructor that takes two parameters, `newItem` and `nextNode`, and populates the `item` and `next` fields accordingly.

```
public Node(Object newItem, Node nextNode) {
    item = newItem;
    next = nextNode;
}
```

Then we have some accessors — that is "setters" and "getters" for setting and getting values of the two private fields.

```
public void setItem(Object newItem) {
    item = newItem;
}

public Object getItem() {
    return item;
}

public void setNext(Node nextNode) {
    next = nextNode;
}

public Node getNext() {
    return next;
}
```

This material should be familiar to you, and you'll no doubt notice some similarities as we work in C++.

Now let's see how we'd code up a "Lecturer" class in C++.

Implementation of `Lecturer` class

(02:45)

As we said, we're going to write a class that represents data regarding computer science lecturers — name, office, and courses taught. This will consist of an implementation of a `Lecturer` class, and a little program to test and verify its function. In doing so, we will learn how to

- write a class in C++ — with the usual constructors and accessors,
- override operators, and
- read data from a file.

Along the way, we'll learn quite a bit about programming in C++.

Let's begin.

In this example, we're going to define our class in a header file. C++ projects are structured a little differently from projects in Java. Generally we put our class declarations — or what's called "the interface" — in a header file with a `.h` extension, and class implementation in another file with a `.cpp` extension. For this course, most of the implementation will go in header files also. This is for a number of reasons: primarily for simplicity's sake and also because we will be using what are called Templates (we'll learn about these later). Templated code is generally implemented in the header file.

So we'll use the header file for class declarations and implementation, and then we'll write programs in other files (with `.cpp` extensions) that make use of the classes thus defined.

(03:36)

Let's start with the header file. We'll call this `lecturer.h`.

(Open new file in IDE for "live" coding.)

Guards

We begin our header file with some preprocessor directives called "guards", these are to make sure that a file in a project isn't included more than once. If you're using CLion, these will be created for you automatically. These values come from the name of the project, "lecturer", and the name of the file "lecturer.h", hence `LECTURER_LECTURER_H` — but there's nothing magical about this. We could use any definition and it would be OK, so long as it were unique within the project. Since there's no reason to change these, let's leave them as they are.

```
#ifndef LECTURER_LECTURER_H
#define LECTURER_LECTURER_H

#endif // LECTURER_LECTURER_H
```

You'll notice preprocessor directives begin with a hash symbol. Now, this guard simply tells C++ that if `LECTURER_LECTURER_H` is already defined to skip everything in this file, otherwise, if this is the first time this file is included, read everything within the guard. All of our code will go within this guard.

Includes

First, we add preprocessor directives to include components of the C++ standard library that we'll be using here. This is much like importing packages in Java, or importing modules in Python. You can guess what many of these are from their names:

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
```

- `fstream` is used for streaming data to and from files. We'll need this, since we'll be reading data from file.
- `iostream` is used for basic streaming I/O.
- `iomanip` is used for manipulating and formatting I/O. We'll use this for formatting output and making tables.
- `string` is for dealing with strings.
- `vector` is for dealing with vectors — that is, arrays. We'll be using a vector to store `Lecturer` records.

So we'll need all of these. You'll see when and where these are being used when we get to the relevant portions of code.

Now on to the class definition.

Class definition

(06:38)

The initial declaration shouldn't surprise you.

```
class Lecturer {

};
```

By convention, we name our classes with initial caps.

Then we have five private fields: `name`, `office`, `course1`, `course2`, and `course3`. The first, name, will hold the lecturer's name.

```
class Lecturer {
private:
    std::string name;
    std::string office;
    int course1;
    int course2;
    int course3;
...
};
```

Office will hold the lecturer's office address. The remaining three fields will hold identifiers for the courses taught by each lecturer. Notice that we've preceded these with a `private:` access specifier. This indicates these fields are private members of the class. Notice also that we prefixed `string` with `std::`. This indicates that we're using the standard string library that was included earlier with `#include <string>`.

Next we have public member methods.

(8:06)

Here we'll use the `public:` access specifier.
Public member methods will include the constructors, setters and getters, and other functions we'll get to a little later.

Let's start with the constructors.

Here we have two. One to handle new objects without any parameters supplied — a kind of default — and the other to construct new `Lecturer` objects when a complete set of parameters are passed in. Here's the first constructor:

```
public:
    // Constructors
    Lecturer() {
        name = "Jane Doe";
        office = "Innovation E200";
        course1 = course2 = course3 = 0;
    }
```

Notice here that we provide default values for `name` and `office`, and we set all three courses taught to zero. Notice the shorthand used to do this.

Next we define a constructor for the more typical case — where we wish to automatically populate our new object with data.

```
Lecturer(std::string name, std::string office,
        int course1, int course2, int course3) {
    this->name = name;
    this->office = office;
    setCourse1(course1);
    setCourse2(course2);
    setCourse3(course3);
}
```

Notice that we have to specify the types of all parameters. What do we do with these parameters? Here we assign the input string — the parameter `name` — to the field `name`. Notice how we do this. What is `this`? What is that arrow?

Every object in C++ has access to itself via a pointer called `this`. `this` points to the object itself and all members can be accessed using `this` and the arrow operator.

So `this->name = name;` can be read as "assign the value of the input parameter `name` to the member field called `name` within the current object." The current object is referred to as `this`.

We do the same thing for `office`.

Now notice that we're going to use setter methods to set the information about which courses are taught by a given lecturer. Why are we going to do this? You'll see in just a moment.

(12:02)

Notice that there are three fields for the courses taught by a lecturer. Each one will have its own setter. Now, the reason we're using a setter is we want to make sure that the course

number can't be negative. We'll see two ways to do this. First, let's implement the setter for `course1`.

```
void setCourse1(int course1) {
    // Do not accept negative course number
    if (course1 < 0) {
        this->course1 = 0;
    } else {
        this->course1 = course1;
    }
}
```

Notice that this method does not return a value so we start the declaration with the keyword `void`. We've given the setter a name `setCourse1` and it takes one parameter — an integer called `course1`. Now what happens? If `course1` is negative we set the value of the field to 0. Notice we use `this` and the arrow operator to access the target member. If the input parameter `course1` is non-negative we assign its value to the field `course1`. That's it.

Now let's look at another way, using the ternary operator. Most languages have a ternary operator, and you may have seen this before. It's just a more compact way of implementing the same behavior as we did in the setter for `course1`.

```
void setCourse2(int course2) {
    // Ternary operator
    // Is course2 < 0? If so, set to 0. If not, set to course2.
    this->course2 = (course2 < 0) ? 0 : course2;
}
```

To the right of the assignment operator, we have a condition `(course2 < 0)` if this is true, we assign the value immediately following the question mark. Otherwise we assign the value following the colon. This is a convenient shorthand, and if you haven't seen it before you should familiarize yourself with it.

Let's do the same thing in the setter for course3.

```
void setCourse3(int course3) {
    // Ternary operator
    this->course3 = (course3 < 0) ? 0 : course3;
}
```

Now let's take care of setters for `name` and `office`, used to update these values outside of

object construction.

```
void setName(std::string name) {
    this->name = name;
}

void setOffice(std::string office) {
    this->office = office;
}
```

That's all there is to these, and we're done with the setters.

(16:17)

The getters are straightforward. We'll use the `const` keyword to indicate that these methods never modify the class's fields.

```
std::string getName() const {
    return name;
}

std::string getOffice() const {
    return office;
}

int getCourse1() const {
    return course1;
}

int getCourse2() const {
    return course2;
}

int getCourse3() const {
    return course3;
}
```

What happened to `this` and the arrow operator? It turns out that we don't need it here. In fact, in most cases it's really only needed when we wish to be explicit or avoid ambiguity. Why did we use it before? Let's go back to some of that code.

```
Lecturer(std::string name, std::string office,
        int course1, int course2, int course3) {
    this->name = name;
    this->office = office;
```

```
    setCourse1(course1);
    setCourse2(course2);
    setCourse3(course3);
}
```

Notice that the variables passed to the constructor — `name` and `office` — have the same names as member fields. So here it's important to disambiguate. So we use `this` and the arrow operator.

Going back to our getters, we see we don't have that problem, so we omit `this` and the arrow operator, and everything works just fine. This is more concise and readable and I think preferred by most C++ programmers (though you'll find some programmers — and workplaces — with different opinions).

Comparison with Java

So now that we've completed the constructors and accessors for this class, let's do a little comparison with `Node.java` before we move on.

(19:14)

For a constructor, in C++ we have

```
public:
...

Lecturer(std::string name, std::string office,
        int course1, int course2, int course3) {
    this->name = name;
    this->office = office;
    setCourse1(course1);
    setCourse2(course2);
    setCourse3(course3);
}
```

and in Java,

```
public Node(Object newItem, Node nextNode) {
    item = newItem;
    next = nextNode;
}
```

(split screen and show highlighted code side by side).

In C++ we use the public access specifier, `public` followed by a colon and then anything that follows has public access. In Java we use the public keyword before each public method in the class definition. But other than that, these are pretty similar.

In the Java constructor, we distinguish between input parameters and member fields by using different names. We could have done the same in C++, by the way, and avoided the use of `this` and the arrow operator. For example, we could have named the parameters `lecturerName` and `lecturerOffice` and that would have worked just fine — avoiding the conflict.

In both languages we need to specify the type of our parameters in the method definition. In both cases, we don't specify the return type because it's a constructor — we know this because the name of the method matches the name of the class.

So there are many similarities between the two languages with regard to the constructors. Now let's look at some accessors.

Here's a setter in C++,

```
public:
...

void setOffice(std::string office) {
    this->office = office;
}
```

and one in Java,

```
public void setItem(Object newItem) {
    item = newItem;
}
```

Notice these are very similar. If our field had been a string in Java we would have indicated this by specifying `String` type. In our C++ setter, we could have renamed the input parameter `office` and avoided the use of `this` and the arrow operator. But it's plain to see there's considerable similarity.

The same holds for getters. Here's a getter in C++,

```
public:
...
std::string getOffice() const {
    return office;
}
```

and here's one in Java,

```
public Object getItem() {
    return item;
}
```

apart from the `const` keyword in C++ — which is not used in Java — these are near identical.

So we see there are a lot of similarities between the two languages — at least in these cases.

Now let's take a break, and in the next video, we'll move on to the rest of our implementation of the `Lecturer` class in C++.