

JCCC Interview

Spring 2018

Lecture 0 - Polymorphism in C++

Learning Objectives

- Learn how **polymorphism** is used to create **class hierarchies**
- Understand the difference between **function overloading** and **function overriding**
- Learn about **virtual functions** and how they can be used in polymorphism
- Learn about **abstract classes** and their uses
- Explore examples of polymorphism

Concepts, Terminology, People

- Polymorphism
- Class hierarchy and Subtyping
- Superclass (parent class or base class)
- Subclass (child class)
- Function overriding
- Protected access specifier
- Virtual functions
- Pure virtual functions
- Abstract classes

Resources

- https://www.tutorialspoint.com/cplusplus/cpp_polymorphism.htm
- <http://www.cplusplus.com/doc/tutorial/polymorphism/>
- <https://www.geeksforgeeks.org/polymorphism-in-c/>
- <https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/>
- <https://stackoverflow.com/questions/5854581/polymorphism-in-c>
- <https://blog.feabhas.com/2010/05/polymorphism-in-c/>
- https://en.wikibooks.org/wiki/C%2B%2B_Programming/Classes/Polymorphism
- [https://en.wikipedia.org/wiki/Polymorphism_\(computer_science\)](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))
- <https://en.wikipedia.org/wiki/Polymorphism>
- [https://en.wikipedia.org/wiki/Category:Polymorphism_\(computer_science\)](https://en.wikipedia.org/wiki/Category:Polymorphism_(computer_science))

Students Should Complete Before Class

- Assignment on Classes in C++

Students Should Start After Class

- Assignment on Polymorphism in C++

Lesson Setup Before Class

- Print activity sheets and cut the objects into separate parts
- Print lesson plan for review and committee
- Prepare examples for compilation on the available systems

Lesson Plan

- Questions / Review
- Lecture
 - Think-Pair-Share Activity
 - Polymorphism Activity
- Assignment

Lecture Notes

1. **[Title]** - Welcome to class! Today we are going to introduce a new topic: polymorphism! Don't worry if you don't recognize that word yet; we'll cover it in a few minutes. Before we start, let's do a quick review to make sure we are on the same page.
2. **[Previously...]** - Previously in this class, we've covered a variety of topics in C++. Hopefully by this point you should be somewhat familiar with them, as they will be very important when introducing this new topic. Are there any questions about these previous topics?
3. **[Classes 1]** - Here's a quick example that covers classes in C++, just to make sure everyone is comfortable. Let's quickly review what it says: **[slide]**

Okay, now that we are familiar with this concept, let's talk about how this might be used in the real world. Let's say we want to develop a program to keep track of information for our computer science department.

Leading Questions:

Can we use this Person class to store that information?

Are there different types of people, with different information, that we should store?

Does it make sense for a Teacher to have a student ID, or a Student to be able to set grades?

How can we resolve those issues?

4. **[Class Hierarchy]** - One way we can do that is by using a hierarchy of classes, to help represent all of the information we need to store. There are some items that every

person will need, but other items that are unique to a student or a teacher. By using a **class hierarchy** (or **inheritance**, as it is sometimes referred to), we can make our code represent how things look in the real world.

Think-Pair-Share - Let's do a quick think-pair-share activity. Take a little time to think of other real-world hierarchies that might be represented in this way. Then, when I tell you to, get with a partner and discuss what you came up with. Finally, I'll call on a few groups to share what they thought of and we'll discuss as a class.

[Timer: 2:00 - 1 min for think, 1 min for pair, then share]

Other examples: Animal Kingdom Classifiers, Food Items, Vehicles, Dewey Decimal System

5. **[Polymorphism]** - Now that we've seen how many useful places this could be used, let's see how we can do that in C++. This is a concept known as **polymorphism**. According to the dictionary, polymorphism is *the ability to assume different forms or shapes*. In programming, we can build objects that are able to assume different forms based on their class hierarchy, and then we can store them and use them in different ways. Let's look at an example
6. **[Classes 0]** - In this example, I have defined 4 classes. You'll notice that after some of the class definitions, I've listed a second class name. That is the **superclass** or **base class** (sometimes called the **parent class**) of the class I'm creating. So, **Person** is a superclass of **Student**, and **Student** is a **subclass** (or **child class**) of **Person**. Notice that a **subclass** of a class can then be a **superclass** of another, creating a multi-level hierarchy, just like we saw in our real-world examples. Likewise, a class can be the **superclass** of more than one **subclass**, as **Person** is in this example.

*Yes, a **subclass** can have more than one **superclass** as well in C++, but we won't cover that today! That's a topic for another time.*

One of the biggest advantages of polymorphism is that any **subclass** can access members in the **superclass**, unless they have been declared private. Likewise, a pointer to any **superclass** type can also be used to point to any of its **subclasses**, so you can have an array of **Person** pointers, but each one points to either a **Student** or **Teacher**.

Let's do a quick example to see how that works: **[Hand out example sheets]**

- a. In groups of 4, give each student a sheet of code and the corresponding object. Leave the 5th object on the table for them to discuss later.
- b. Ask all of the students to look at their code and object, and see if they know how to do each of the functions defined on their sheet: **happyBirthday**,

changeMajor, promotion. Hopefully they should.

- c. Now, have them hand their object to their **superclass**. Ask the question again, and see if they understand. The student with the **Teacher** code should now have the **Administrator** object, and the student with the **Person** code should have both the **Student** and **Teacher** object.
- d. Repeat once again. The student with the **Person** code should now get the **Administrator** object.
- e. Now, for some gotcha's: hand the **Student** object to the person with the **Teacher** code, and hand the **Teacher** object to the person with the **Student** code. Hand the **Person** object to the student with the **Administrator** code.

Could they still do their operations? Yes!

Could this be a problem! Yes!

How do we prevent objects from being used by anything other than their superclasses? The compiler enforces type-checking to prevent this! (But, if you are clever, there are still ways you can cause problems!)

- f. Look at the unlabeled object.

Can we decide which class it belongs to? No! We must tell the compiler which class each object should be when it is created, or else we have no way of knowing what it should be. (We'll see later how we can use the `dynamic_cast` and `typeid` operators to determine what type an object is and how to change types dynamically)

- 7. **[Example 1]** - Now let's take a look at some examples in code and see if we can figure out how they work. This first example is pretty simple - we just have a **Person** object, and we use the **age** and **happyBirthday** members. **What should be the output here?**
- 8. **[Example 2]** - Here's another one. Now we have a **Student** object, which we are also pointing to with a **Person** pointer. **What should be the output here?**
- 9. **[Example 3]** - Here's yet another using an **Administrator** object with **Teacher** and **Person** pointers. **What should be the output here?**
- 10. **[Subclass]** - As we've discussed, a **subclass** can access any **superclass** members which are made public. As we learned when we introduced the **private** access modifier, those members cannot be read by any other class. What if we'd like our **subclasses** to be able to access some members, but **NOT** anyone else? In that case, we can use a new access modifier, **protected**.
- 11. **[Classes 1]** - Here's an example of that. Now we've made the **age** and **happyBirthday**

members **protected** in **Person**. I've also added a few new members to **Student** to help us account for that.

12. **[Example 4]** - Let's look at this example. **What will happen with this code?** In this case, the compiler will actually prevent compilation of the code, because in the **main** method we are trying to directly access **protected** items from **Person**. **Can you see why that is?**

How can we modify this code so that it works? What new members should we use instead? (Introduce getters and setters) See how we can access those items by using public members of Student?

13. **[Example 5]** - Here is the fixed code for that example. **What should it output this time?**

14. **[Subclass]** - Let's go back and review what we said earlier. Now, you can see that a **subclass** can access any **public or protected** members in its **superclass**.

Let's ask another useful question: **can we have the same function in a superclass and its subclass, but make them have different code / do different things? (What is a use-case for this?) Yes we can!**

15. **[Classes 2]** - Here's an example of that. You'll notice that we have a **whoAmI** function in both the **Person** class and its subclass, **Student**.

16. **[Example 6]** - Let's consider this code. **What should the output be?**

17. **[Example 7]** - What about this example, where we have a **Person** pointer pointing to a **Student** object. **Which one should it use this time?** (This one is tricky!)

18. **[Overriding]** - This is an example of **function overriding**. Recall that **function overloading** is having multiple functions with the same name but different **function signatures** (the number, order, and types of parameters). This is a very important aspect of **polymorphism**, as it allows us to have objects with the same function names, which do different things.

However, many of you might have noticed a problem here. **What is it?**

What if we want the object to call the function based on the type it actually is, not the type it appears to be? We can use the **virtual** keyword for that.

19. **[Classes 3]** - Here's an example of how that would work. Notice here I've declared the **whoAmI** function to be **virtual** in the **Person** superclass.

20. **[Example 8]** - Now, if we redo this example, **what should happen here?** It will be the

same as before, since each object was declared to be different objects at the beginning.

21. **[Example 9] - What about this example?** In this case, we have a **Student** object, and a **Person** pointer. Since the **Person** pointer is pointing to a **Student** object, and the **whoAmI** function is declared to be virtual, it will call the function in the **Student** object. **THIS IS THE COOL PART OF POLYMORPHISM!**

How could we use this to our advantage? (*Store arrays of the superclass, have each one do operations based on the subclass type using the same functions*)

22. **[Virtual Functions]** - As you can see, **virtual functions** are functions which can be **redefined** by the subclasses. When a virtual function is called, it uses the code in the **originally created class**, not the **apparent class** based on the current pointer.

Do we always have to implement a virtual function in the superclass if we never intend to use it? No! We can make it an *abstract class* with *pure virtual functions*.

23. **[Classes 4]** - Here's an example of what that looks like. Notice that assign the declaration of the **whoAmI** function in the **Person** class to 0. This will make it a **pure virtual** function. Any class containing at least one **pure virtual** function is called an **abstract** class. It comes with a caveat, as we'll see in the next example.
24. **[Example 10]** - Consider this example. **What will happen when I use this code?** The big caveat with **abstract** classes is that they cannot be directly instantiated, because there is no code for the **pure virtual** function it contains. The compiler will throw an error.
25. **[Example 11]** - Here's another example. Notice that we are not instantiating a **Person** object, just a pointer to one. **What will this code do?**
26. **[Review]** - Here's a quick review of what we've covered today: **[slide]**
27. **[Next Steps]** - After this, we can start to introduce many new concepts which are very important to **polymorphism**, such as these: **[slide]**
28. **[Assignment]** - To get some practice working with **polymorphism**, **virtual** and **pure virtual** functions, and **protected** members, here's your assignment. **[Review Assignment Sheet]**

After Lecture Activities

- *<discuss assignment>*