Lab 4 - All About Inheritance (And Other Stuff Too)

Goals

- In this lab, you will experience:
    - Basic inheritance
    - Multiple inheritance
    - The diamond problem and ambiguity in method calling
    - Composition vs. inheritance
    - A sneak peek at polymorphism

Forewarning

- **This lab does not follow the usual folder structure from the last labs.** This lab is structured into 7 independent tasks, where each task correspond to a source file.
    - **We recommend you split the work amongst your teammates.**
    - Remember to still include **team.txt** in the project root folder, without putting it within a **doc/** directory (look at the folder structure in the Submission section). If you do not, you and all of your group members are at risk of not getting credit! So please make sure to be careful and follow the submission instructions carefully.
- Each program should complete its run with no errors, with its last line printing "**SUCCESS**" once you have completed a program successfully, without any segmentation faults or failed assertions.
- You can run **make** in order to test all programs, or type in the number prefix of the source file name after make (etc.) in order to test a task (e.g. **make 02** will test **02-protected-vs-private.cpp**).

Task 1 - Where's My Parent? (01-wheres-my-parent.cpp)

- There are two classes: Mammal and Human.
- Human wants to use a Mammal method by inheriting from it.
    - But, Human does not currently inherit from Mammal.
- Your task is to make it so that Human inherits with public access from Mammal.
    - This should be a relatively easy task -- it requires 1 line of code.

Task 2 - Protected vs. Private (02-protected-vs-private.cpp)

- There are two classes: InnerCityRing and OuterCityRing.
- OuterCityRing inherits from InnerCityRing. But it cannot access the private variable **inner_city_population** from within itself.
    - Why not? There are multiple solutions, but you don't want to make it public.
- You task is to make it so that **inner_city_population** can be accessed by a child class by changing its access specifier.
    - The **inner_city_population** access specifier should be neither public nor private.
    - This should be a relatively quick task -- it requires changing 1 word within the code.

Task 3 - The Diamond Problem (03-the-diamond-problem.cpp)

- When a class inherits from two parents that contain a member function with the same type signature and same name, the child will have to explicitly specify which one they want to call.
    - This ambiguity between the name/type signature because two of them exist is called **the diamond problem.**
        - You should seek to avoid the diamond problem when you design classes using inheritance in the future.
        - This is the reason why multiple inheritance can be harmful if abused.
- In this task, class D inherits from B and C, where B and C both inherit from A.
- Your task is to make it so **D.do_the_thing()** will call C's version of **print()** instead of B's version of **print()**.
- Side note: This is probably your first exposure to **std::stringstream** and **std::ostream&**. You don't need to know this, but it would be nice for you to know. Note that **std::stringstream** inherits from **std::iostream** which inherits from some form of **std::istream** and **std::ostream**. So, we can use **std::stringstream&** as a **std::ostream&**.

Task 4 - Overpowered RPG Character (04-overpowered-rpg-character.cpp)

- Derived classes can call the constructor of their base classes in their initialization list.
- Here, you will implement two derived classes -- Wizard and Goblin -- that provide fixed values for their base class constructors.
- Your task is to implement the classes Wizard and Goblin such that:
    - They both have a constructors that accepts only **std::string** that becomes the Unit.name of each class.
    - For Wizard:
        - health = 1000
        - mana = 100
        - attack = 10
        - defense = 0
    - For Goblin:
        - health = 100
        - mana = 10
        - attack = 10
        - defense = 10
    - For Wizard, you will also implement another member function with the signature:
        - **void blizzard(Goblin& other)**
        - ...that will change the `other` **Goblin**'s defense to 0.
    - This task looks like it has a lot of code, but you won't have to touch any of the existing code -- you just need to design the 2 new classes that inherit from Unit.

Task 5 - Composition vs. Inheritance (05-composition-vs-inheritance.cpp)

- **Composition**, in contrast to inheritance, is when objects can own (or be composed of) other objects. You've already used this in Lab 2 when **Person** contained **Point2D**.

- In this file, **Wizard** inherits from **Human** and **MagicStaff**.
    - However, this is a "code smell:" a Wizard is a Human but should *own* a MagicStaff, not *be* a MagicStaff.
        - It would make sense to use composition for MagicStaff, and inheritance for Human.
- Your task is to change the class Wizard such that it no longer inherits from MagicStaff, but instead owns or "has" a MagicStaff that it uses to **cast_fireball()**.
    - i.e. make Wizard use composition with MagicStaff instead of inheritance
- Side node: The code to check whether Wizard can call **fireball()** directly as a public member function since it inherits from MagicStaff currently is a bit complicated -- it uses something called SFINAE with templates in order to perform the check at compile-time. This is a technique from what is called C++ template metaprogramming. You are not expected to know this at this point (if at all).

Task 6 - Virtual Functions (06-virtual-functions.cpp)

- Child classes can overload methods from parent classes.
- Sometimes, we want to be able to treat child classes the same as its parent type:
    - e.g. "All Mammals have **walk()**" or "All Shapes have an **area()**"
- We want to use a parent pointer to refer to the address of child objects *and* when you call member functions using the parent pointer, it will automatically look up the actual type of the child object and call the child member functions. **See Figure 1.**

```cpp
#include <iostream>

class Mammal {
public:
  virtual ~Mammal() {}
  virtual void print() const {
     std::cout << "Mammal!" << std::endl;
  }
};

class Whale : public Mammal {
public:
  virtual ~Whale() {}
  virtual void print() const override { // this line is important for syntax
    std::cout << "Whale!" << std::endl;
  }
};

int main() {
  Whale whale;
  whale.print();
  Mammal mammal;
  mammal.print();
  Mammal* ptr_mammal = new Whale();
  ptr_mammal->print();
  delete ptr_mammal;
  return 0;
}
```

Figure 1 - A small example program that shows virtual functions being used to implement what is called polymorphism. The Mammal* ptr_mammal, because it points to a Whale object, will call the Whale virtual member function when it is used to call **print()**. If the **print()** function were not virtual, it would call the Mammal member function always since it's just a regular Mammal*.

- To be able to use the same member function on the parent to call the specific versions on the children using a parent pointer, we can mark the member function as **virtual** and provide **override** for the child's class member function that "override" the functionality of the parent's.
- Don't worry, you won't need to understand this too much yet. You just need to make sure that you can change regular member functions into virtual member functions.
- Your task is to convert **print()** on A and ChildA into virtual functions.
    - If you have trouble remembering/knowing the syntax, please take the initiative to look up the syntax to perform such a thing.

Task 7 - Sneak Peek [of] Polymorphism (07-sneak-peek-polymorphism.cpp)

- Read the intro for Task 6 (see above).
- **Pure virtual functions** are virtual functions that delete their default functionality, but still leave themselves open to virtual function overloads from child/derived classes. (I guess you could think of them as "pure" because they are "purely virtual" which means they don't have any "real functionality" by themselves.)

    - Classes that inherit from classes that have pure virtual functions *MUST* implement/overload them if they are to be constructable (or, basically used in a program).
    - They're used to mark entire classes as **abstract classes**, such as "Shape" or "Money," which makes them non-constructible.
        - You can't "construct/touch" an abstract concept like "Shape" or "Money," but you know of specific instances such as "Rectangle" or "Euro."
- Abstract classes are used to implement **interfaces**, or "common behaviors that a category of objects should have/implement." In C++, we do this by creating an abstract base class and have objects that conform to an interface inherit from them and
    - By convention, people may often mark their "interface" classes with an "I" before the rest of the name as a shorthand: for example, "Interface for Printable objects" ⇒ IPrintable
    - e.g.
        - IPrintable has **print()**
        - IAreaable has **area()**
        - IRender has **render()**
- Your task is to implement a class named **Zap** that inherits from the abstract class called **IMagicSpell**, and overrides **cast_spell()** similarly to the other classes to print "Zap!" on its own line when that function is called.
    - There's already 2 other classes that have very similar implementation patterns that you can take cues from quite closely...

Submission

- **Comment your code! If your code does not work correctly, you may be able to get *partial credit* based on your thinking that we can infer from the comments!**
- In a file called **team.txt**, write your group number on the first line. On subsequent lines, write the last, then first names of your team members separated by a single comma, each on its own line. If your name contains whitespace (e.g. the single space in the last name, "da Silva"), you may keep the whitespace.
- Place your source code files into a directory called **project**, using the following directory structure:
    - `project/`
        - `Makefile`
        - `<your modified .cpp source files>`
        - `team.txt`
- Archive and compress your **project** directory using the command:
    - `tar czf name_of_archive_file.tar.gz project`
    - The name of the archive file should be in the following format:
        - `ee205-lab<lab-number>-group<group-number>.tar.gz`
- Submit the *tar.gz* file under your group's entry on Laulima for the assignment.
    - For your safety, test uncompressing and unpacking your archive in a clean folder, then compile your source files. Ensure that the executable will run correctly.

Supplementary Material

Inheritance vs. Composition

- Preposition: Inheritance vs. composition is complicated, and is worthy of study.
- *Inheritance* is a powerful feature. In object-oriented programming, inheritance is even sometimes put down as worse compared to *composition*. (The classic book on OOP design patterns, written by the "Gang of Four" (GoF), says to *prefer* composition over inheritance. But why?)
    - **Inheritance** - classes can inherit properties automatically from "parent" classes.
        - E.g. A **Dog** is a **Mammal**
    - **Composition** - objects can be composed up of other objects.
        - E.g. A **Car** has a **Tire** (and hopefully 3 more)
- Notice: *is-a* vs. *has-a* relationship. Sometimes, it is very easy to distinguish this pattern (e.g. **DefaultOptionDropdownList** is-a **DropdownList** with a default option). However, it can be hard:
    - Ex: If you have a **Boy** class, should a **Wizard** class inherit from Boy? If so, does that it mean it should also inherit from **WizardHat** because all **Wizard** objects should have a hat?
        - Is it correct to say that **Wizard** = **Boy** + **WizardHat**?
        - Wait, should the class be called **BoyWizard**?
        - Perhaps: "A wizard is-a boy that has-a wizard's hat."

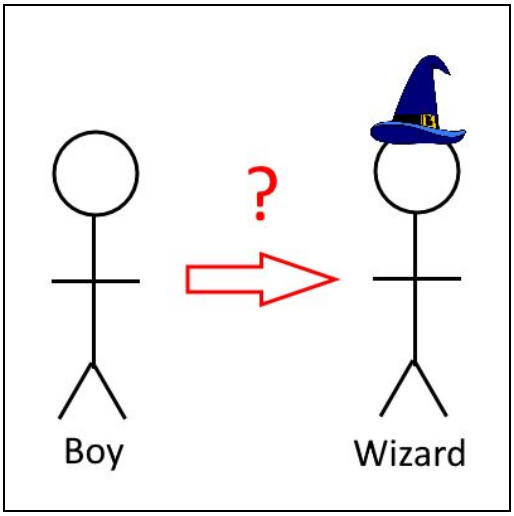

Figure 2 - How might a **Boy** class be related to a **Wizard** class if a Wizard must have a **WizardHat**?

    - Ex: If you have a **Bird** class that has a member function called **fly()**, how can **Penguin** should inherit from **Bird** but be unable to fly?
- Conclusion: object design is a complicated issue, and is worthy of study.

Types and Inheritance

- In computer science, type systems are ubiquitous. Type systems are essentially "in-built proofs" that a certain "object" satisfies certain properties -- for example, in C++, the **int** type demands that you can perform arithmetic on it. Despite your computing representing all data as a series of binary-formatted data in memory, your program is written with the idea that "certain bits have different flavors," and that some of them don't go well together (e.g. you can't directly invoke `int** + std::vector<int>`). This is the power of types -- with type comes context, and safety through "proven" context.
- When you define new classes, you are defining new object types within C++. Thus, you can think of new classes as extending the type system with your own types. You're a type wizard!
    - Therefore, when you use inheritance, you are relating two types together: a **Boy** is-a **Wizard**, or that "all Wizards are Boys with possibly different properties."
        - Did you really want to restrict Wizards to be Boys by default?

Interfaces

- Interfaces represent *capabilities* that classes that *implement the interface* might provide:
    - Ex:
        - **Iterable** (represents an iterator, or an object designed to encapsulate how to "step through" a collection of items like an Array or Vector)
        - **Printable** (the object implements the "string output" operator)
        - **Killable** (e.g. for a game that wants certain "units" to be "killable")
- ***One-liner:*** interfaces force inheriting (or *implementing*) classes to *implement* their functionality.
- Interfaces are conceptually different than classes. However, in C++, you can create an interface "by convention" using classes and inheritance.
- In C++, interfaces are "classes" that have the following properties:
    - **No state** (no variables)
    - Cannot be created (or, **no *instantiation*** as a runtime object)
        - In other words, you can have pointers to an interface type, but not create one directly. This is related to **polymorphism**.
    - **Define required functions** for those that inherit from it
- In C++, you can use an **abstract class** to do this.
- Interface classes usually have peculiar naming conventions:
    - Since Interface starts with the letter I, we can stick it in front of the names of our abstract classes to create a convention to signal to readers that they are interfaces: IIterable, IPrintable, and IKillable.

    - Additionally, since interfaces usually define behavior that an object can adhere to, usually it takes the form: **<verb>-able**.
        - E.g. Enumerate ⇒ IEnumerable
        - Of course, there are always exceptions when a word doesn't sound well, such as "things that act like a magic hat" ⇒ IMagicHat.

- **Why should I use them?**
    - Opt-in functionality -- you can reason about objects by the interfaces they adhere to
    - Compiler ensures that objects *must* at least *implement* the interface's member functions because they're pure virtual functions.

The Diamond Problem

- Tip: draw a diagram for this part. Try using boxes and arrows using the convention here with the picture on the right for the arrows.
- Inheritance is usually implemented as a class hierarchy. Think: a family tree.
- Suppose you are creating a some sort of video game or simulation. It has a **Mammal** class that is inherited by **Hooman** (pronounced "hoo-man") and **Animan**. They all implement the member function **speak()**.
    - What happens when you want to create a **Hybrid** class that inherits from both **Hooman** and **Animan**? Does it invoke **Hooman.speak()** or **Animan.speak()**?
- The diamond problem is when the class hierarchy diverges and then converges again, creating "ambiguity" in member function calls because more than one parent version of the call can be valid. Think: inbreeding in the class hierarchy. Any invocations of **Hybrid.speak()** where **speak()** is not directly defined for **Hybrid** will now have to choose between:
    - **Mammal.speak()**
    - **Animan.speak()**
    - **Hooman.speak()**
    - So, when you call a member function on, say, a **Hybrid** object with the name **"**hybrid":
        - hybrid.Mammal::speak()
- Think: if you are designing a drink dispenser, upon asking for "Water," would you ever design a machine that would:
    - Stop the customer
    - Ask explicitly if they wanted:
        - HotWater
        - ColdWater
        - CarbinatedHotWater
        - Or "Water?"
            - ...everytime they wanted to get Water? No.
- The diamond problem is exposed here because C++ sports a multiple inheritance system.