CSCI 3081W Workshop 2 - Inheritance, Polymorphism, and Abstract Classes

Spring 2025

Submission:

After you complete your code, save it as `workshop2.cc` and submit it on Canvas/Gradescope under the Workshop 2 assignment. Workshop 2 is due Thursday, February 13th at 11:59 pm.

If you worked with someone on this workshop, comment their name and x500 at the top of your file before your include tags. Alternatively, you can submit as a group in Gradescope.

Example:

```
// Shonal Gangopadhyay gango010
// Kevin Wendt wend0144

#include <iostream>
…
```

Goal:

The goal of this workshop is to practice translating design documents into code by implementing classes and relationships defined by UML Class diagrams. You will also gain experience with abstract classes and the concepts of inheritance and polymorphism, as well as testing and debugging code implementing hierarchical structures.

Prerequisites:

Be able to SSH into a lab machine via VSCode or your terminal, and make sure your quota is below 1.5GB.
https://github.umn.edu/umn-csci-3081w-s25/FAQ
~or~
Know how to use vole, and make sure your vole quota is below 1.5GB.
(Else, contact csehelp@umn.edu)
https://cse.umn.edu/cseit/self-help-guides/virtual-online-linux-environment-vole

Overview:

This workshop contains two sections: `OrderMethod` and `HardwareComponent`.

Food Ordering System:

In the first part of this workshop, you will explore how inheritance and polymorphism work in C++ using the concept of different methods for placing an order at a fast food restaurant (e.g., Dine-In, Takeout, Drive-Thru, Delivery). First, you will develop a set of classes without polymorphism enabled. Then, you will test the functionality of the system. Lastly, you will enable polymorphism in your classes and test the system to see how behavior changes when using inheritance.

You will implement a class hierarchy where the `OrderMethod` class acts as the base class, and the `DineIn`, `Takeout`, `DriveThru`, and `Delivery` classes act as derived classes that inherit from `OrderMethod`.

Your task is to:

1. **Develop the base class** `OrderMethod` and its derived classes `DineIn`, `Takeout`, `DriveThru`, and `Delivery` without polymorphism.
2. **Test the functionality** by calling methods specific to each ordering method.
3. **Modify the classes** to enable polymorphism using the `virtual` keyword and observe how behavior changes when using inheritance.

Computer Hardware:

In the second part of the workshop, you will explore **inheritance, polymorphism, and abstract classes** in C++ using a computer hardware simulation as an example. You will implement a class hierarchy where the **`HardwareComponent`** class serves as the abstract base class, and specific hardware components like **`CPU`**, **`GPU`**, and **`RAM`** inherit from it.

You will:

1. **Implement the abstract class** `HardwareComponent` and its derived classes (`CPU`, `GPU`, and `RAM`).
2. **Use the concept of pure virtual functions** to enforce that all derived classes must implement certain functionality.
3. **Test how this structure enables polymorphism** by working with hardware components through base class pointers.

You will write all of your code in a single .cc document (probably 100+ lines). I know that usually you would have a set of .cc and .h files, but it's really hard to grade if you do that.

How to Compile and Run:

Compile: `g++ workshop2.cc`
Run: `./a.out`

Compile: `g++ workshop2.cc -o workshop2`
Run: `./workshop2`

Frequently recompile and test your program as you develop!

Food Ordering System

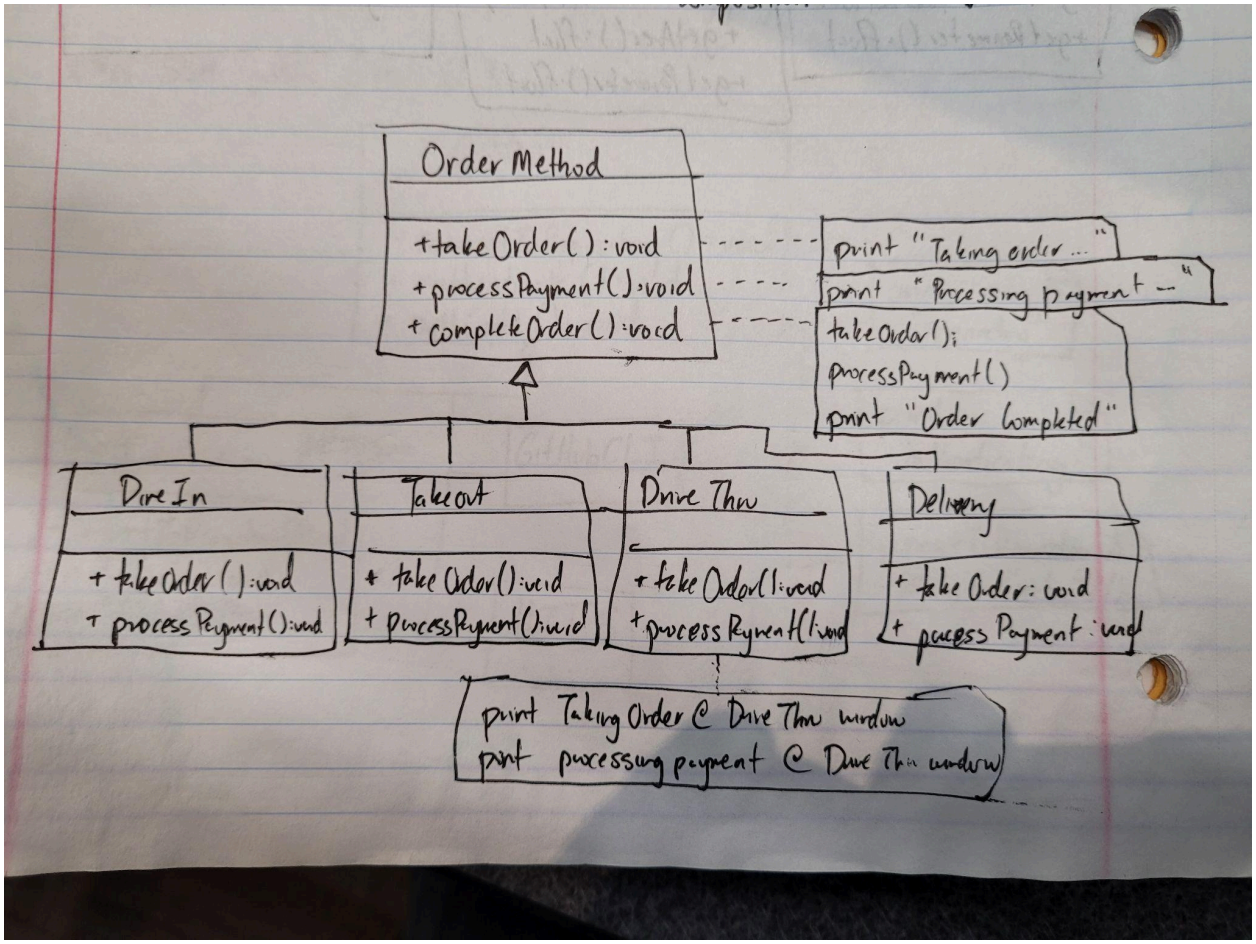
Part 1: Implementing Ordering Methods Without Polymorphism

Step 1: Building the Base Class (**`OrderMethod`**) and Derived Classes (**`DineIn`**, **`Takeout`**, **`DriveThru`**, **`Delivery`**)

In this part of the workshop, you will implement a class hierarchy representing different ways to place a food order at a fast-food restaurant. The `OrderMethod` class will serve as the **parent class**, and the `DineIn`, `Takeout`, `DriveThru`, and `Delivery` classes will inherit from `OrderMethod`.

- **Inheritance:**
  Inheritance in C++ allows one class (the **derived** class) to inherit properties and behaviors (**methods**) from another class (the **base** class). You'll use this feature to define how different ordering methods (Dine-In, Takeout, Drive-Thru, and Delivery) share **similar behaviors** (like taking an order and processing payment).

- **UML Structure:**
  Here's a UML diagram for the structure:



Class Descriptions:

- **Base Class (`OrderMethod`)**:
  This class represents the general behavior of placing an order at a restaurant. It contains methods common to all ordering methods:
  - `takeOrder()`: Simulates taking an order.
  - `processPayment()`: Handles payment processing.
  - `completeOrder()`: Combines taking the order and processing payment before finalizing the order.
- **Derived Classes (`DineIn, Takeout, DriveThru, Delivery`)**:
  Each derived class represents a specific ordering method and **overrides** inherited behaviors:
  - `DineIn`: The customer orders at the counter and eats inside the restaurant.
  - `Takeout`: The customer picks up the food and leaves.
  - `DriveThru`: The customer orders and receives food from a drive-thru window.
  - `Delivery`: The restaurant delivers the food to the customer's location.

Step 2: Testing the Base and Derived Classes

Write a `main()` function that tests the functionality of the classes. Here's an example test case you can use:

```
int main() {
    OrderMethod* order;

    // Test Dine-In
    cout << "Dine-In Order:" << endl;
    order = new DineIn();
    order->completeOrder();
    delete order;

    // Test Takeout
    cout << "\nTakeout Order:" << endl;
    order = new Takeout();
    order->completeOrder();
    delete order;

    // Test Drive-Thru
    cout << "\nDrive-Thru Order:" << endl;
    order = new DriveThru();
    order->completeOrder();
    delete order;

    // Test Delivery
    cout << "\nDelivery Order:" << endl;
    order = new Delivery();
    order->completeOrder();
    delete order;

    return 0;
}
```

Step 3: Observing Output

Run your program. Notice that the base class implementation of `takeOrder()` and `processPayment()` is being used in all cases. This is because polymorphism has not been enabled yet.

Part 2: Enabling Polymorphism

Step 1: Introducing the `virtual` Keyword

To enable **polymorphism**, we need to modify our base class to allow derived classes to **override** its methods. Polymorphism allows functions to behave differently based on the object calling them.

Modify the `OrderMethod` base class methods to be virtual, allowing derived classes to override these behaviors:

```
virtual void takeOrder() {
    cout << "Taking order..." << endl;
}

virtual void processPayment() {
    cout << "Processing payment..." << endl;
}
```

Then, in each derived class (`DineIn, Takeout, DriveThru, Delivery`), override these methods to provide specific behavior.

For example:

```
class DriveThru : public OrderMethod {
public:
    void takeOrder() override {
        cout << "Taking order at the drive-thru speaker..." << endl;
    }

    void processPayment() override {
        cout << "Processing payment at the drive-thru window..." <<
endl;
    }
};
```

Step 2: Testing Polymorphism

Re-run your test case from Part 1. Now that you've added virtual to the base class methods, the correct method implementations from the child classes will be called.

For example, when ordering `Drive-Thru`, you should see:

```
Taking order at the drive-thru speaker...
```

```
Processing payment at the drive-thru window...
Order completed!
```

This works because polymorphism ensures the correct version of `takeOrder()` and `processPayment()` from the **derived class** is executed, even when accessed through a **base class pointer**.

By adding polymorphism to your fast food ordering system, you've made it possible for each ordering method to implement its own specific behavior while still sharing a common structure.

Hardware Component

Step 1: Defining the Abstract Class `HardwareComponent`

What is an Abstract Class?

An **abstract class** in C++ is a class that cannot be instantiated directly. Instead, it serves as a blueprint for other classes. You create an abstract class when you want to define a common interface for a group of derived classes but leave certain functions unimplemented in the base class, expecting the derived classes to implement them. This is done using **pure virtual functions**.

- **Pure Virtual Functions**: A pure virtual function is a function declared in the base class but not defined there. By marking a function as pure virtual, you enforce that any derived class must provide its own implementation. In C++, a pure virtual function is declared by adding = 0 at the end of the function declaration, as shown below:

  ```
  virtual void getSpecs() = 0;
  virtual float getPowerConsumption() = 0;
  ```

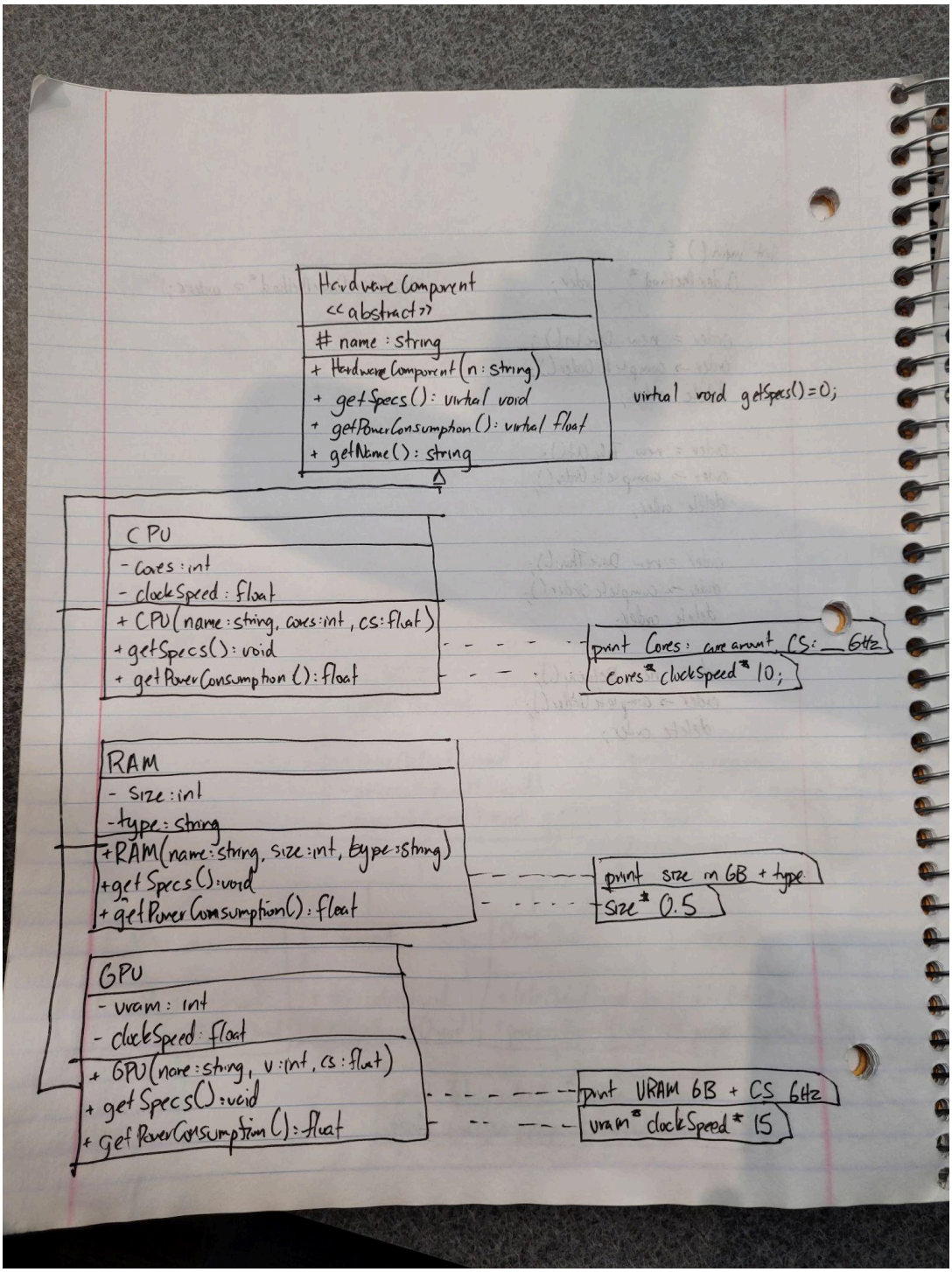
- This means that `HardwareComponent` cannot provide an implementation of `getSpecs()` or `getPowerConsumption()`, but every derived class (e.g., `CPU`, `GPU`, `RAM`) must define how these functions behave.

Class Overview:

- **Base Class (`HardwareComponent`):**
  This class will define the common properties and functions of all hardware components. It will include:
  - `getSpecs()`: A **pure virtual function** that forces each derived class to define how to display its specifications.
  - `getPowerConsumption()`: A **pure virtual function** that forces each derived class to define its power consumption.

- `getName()`: A **non-virtual function** that returns the name of the component.

- **UML Structure:**



Step 2: Implementing the Derived Classes (`CPU, RAM, GPU`)

Class Descriptions:

- **Base Class** (`HardwareComponent`)
  This class serves as the abstract base class for hardware components. It has the following methods:
  - `getSpecs()` (pure virtual): Each component implements its own specifications display.
  - `getPowerConsumption()` (pure virtual): Each component implements its own power consumption calculation.
  - `getName()`: Returns the name of the hardware component as a string.
- **Derived Class** (`CPU`)
  - Inherits from `HardwareComponent`.
  - Implements the `getSpecs()` function.
  - Implements the `getPowerConsumption()` function.
  - Stores the number of cores and clock speed as member variables.
- **Derived Class** (`RAM`)
  - Inherits from `HardwareComponent`.
  - Implements the `getSpecs()` function.
  - Implements the `getPowerConsumption()` function.
  - Stores the memory size and type as member variables.
- **Derived Class** (`GPU`)
  - Inherits from `HardwareComponent`.
  - Implements the `getSpecs()` function.
  - Implements the `getPowerConsumption()` function.
  - Stores the VRAM size and clock speed as member variables.

Step 3: Implementing and Testing the Classes

Implement the base class and its derived classes. Once you're done with that, add this to your main program to test out your code:

```
vector<HardwareComponent*> components;
components.push_back(new CPU("Intel i9", 12, 3.8));
components.push_back(new RAM("Corsair 32GB RAM", 32, "DDR5"));
components.push_back(new GPU("NVIDIA RTX 3080", 10, 1.7));

for (HardwareComponent* c : components) {
    cout << "Component: " << c->getName() << "\n";
    c->getSpecs();
```

```
    cout << "Power Consumption: " << c->getPowerConsumption() <<
"W\n\n";
}
```

In this section of the workshop, you learned how to create an **abstract class** with **pure virtual functions**. This design ensures that every derived class implements specific functionality while sharing common behavior through inheritance. You also observed how **polymorphism** allows a base class pointer to call the correct derived class functions at runtime, demonstrating the flexibility and power of object-oriented programming in C++.

[Optional] Extra Credit - Overclocking via Multiple Inheritance

What is overclocking? Don't overthink it. For this context, we'll define overclocking as increasing the performance of a hardware component beyond its default specifications.

In C++, you can define an interface that allows different types of hardware to share a common behavior without modifying their base implementations. With this in mind, think about how you might enable certain components to be overclocked without altering their fundamental design.

Your goal is to implement an overclocking system where certain hardware components can be overclocked while others cannot. You'll need to design a way to enable overclocking without directly modifying every individual hardware class.

Use what you know about inheritance and think carefully about how overclocking might affect a hardware component's attributes. If you need help, feel free to look up resources on interfaces, multiple inheritance, and virtual functions.

HINT: Have `CPU` and `GPU` inherit from both `HardwareComponent` and `Overclockable`. The `Overclockable` interface ensures that `CPU` and `GPU` can both implement `overclock(float percentage)`. Overclocking increases the clock speed dynamically. `getPowerConsumption()` adjusts based on the new clock speed.

If you choose to finish this and are able to test its functionality correctly in your main function, your testing must be super obvious that it works when I look at it, so make sure to have plenty of print statements to prove that it works.

Additionally, leave a comment "extra credit attempt - <names here>" with anyone you worked with on the extra credit. If you did it solo, leave the comment "extra credit attempted - no partners".

Example:

```
// Shonal Gangopadhyay gango010
// Kevin Wendt wend0144

// extra credit - Shonal only

#include <iostream>
…
```