Griffin Belcher

```yaml
course: OOP (Object-Oriented Programming)
artifact: Mega Worksheet + Study Guide + Rubric
coverage:
  - Stacks & Queues (ADT mindset refresher)
  - Overloading vs Overriding
  - Constructors & Initialization Lists
  - struct vs class
  - Operator Overloading
  - friend
  - Composition (intro, not yet the full assignment)
due:
  day: Monday the 16th
  time: 11:00 AM
in_class_support:
  day: Wednesday
  note: Questions encouraged. Panic discouraged.
tone: Academically serious, emotionally sarcastic
philosophy: |
  OOP is not about inheritance.
  It is about designing types that are hard to misuse.
```

🧠 OOP Worksheet & Study Guide

Making Your Own Types Feel Like the Language Meant Them to Exist

Due: 16 Feb 2026 by class time.

> If this feels long, good.
> If it feels unfinished, also good.
> If you feel like there's "always more," congratulations — that feeling is the curriculum.

This document serves **three purposes**:

1. Homework
2. Study guide
3. A mild personality test for how you think about software design

Read the questions carefully. Many are intentionally phrased to trip up *rote memorization*.

Part 0 — A Necessary Reality Check

Most people think:

> **OOP = inheritance**

> **Math = long division**

Inheritance exists.
It is sometimes useful.
It is wildly overused.

This worksheet is about **building abstract data types (ADTs)** that behave like they belong *in the language* — not about summoning class hierarchies like Pokémon.

Part 1 — Review, But With Consequences

1️⃣ Stacks & Queues (ADT Reality Check)

Answer **clearly and concisely**.

1. Compare **array-based** vs **list-based** implementations of stacks and queues:
   - memory layout — arrays are faster to access, but have limited size
   - resizing behavior
   - cache friendliness — lists vary in size, are easy to insert/remove fro... but have more overhead
   (Yes, cache friendliness matters. No, you may not ignore it.)

2. Why is `std::vector` a natural fit for a **stack**, but awkward for a **queue**? — vectors can only push/pop the back of the vector; queue requires pop at... the front of the queue
3. Define the **invariant** for:
   - a stack — data is inserted and removed from the "top" of the stack
   - a queue — data is removed from the "front" and added to the "back"

> If your invariant takes more than one sentence, it's not an invariant — it's a confession.

Part 2 — Overloading vs Overriding

(Same Word Root, Completely Different Beasts)

2️⃣ Conceptual Distinction

Fill out the table:

Feature	Overloading	Overriding
Resolved at	? compile	? runtime
Requires inheritance	? no	? yes
Same function name	? yes	? yes

Feature	Overloading	Overriding
Polymorphism involved	? Yes	? Yes

Then answer:

1. Why is **overloading** a *compile-time convenience*? Because they are different methods with the same name
2. Why is **overriding** a *runtime contract*? Because what is running is based on the object
3. Why do beginners confuse the two? The names are very similar, and so are their uses
4. Why is that confusion dangerous? Because it can introduce dificult to find bugs

> Hint: The compiler is not your therapist. It will not guess your intent.

Part 3 — Constructors & Initialization Lists

(Where C++ Stops Holding Your Hand)

(3. Widget(int x, std::string y) :: x(id), y(name)

3 Initialization Lists or Else

Given:

```
class Widget {
private:
    const int id;
    std::string name;

public:
    Widget(int id, std::string name);
};
```

Answer:

1. Why **must** this constructor use an initialization list? Because id is a const
2. What happens if you try to assign `id` inside the constructor body? The program has an erro messa
3. Write the correct constructor.
4. Name **one other situation** where initialization lists are required (research-lite). Initialization lists are required when a data member is a const data type

> Saying "because the compiler told me to" is not an explanation.

---

4 Copy Constructor vs Assignment Operator

Research + reasoning required.

1. When is the **copy constructor** invoked? When the object is created
2. When is the **assignment operator** invoked? When a value is assigned to an existing object

> "They both copy stuff" earns partial credit and a sigh.

Part 4 — `struct` vs `class`

(Same Machine Code, Different Intent)

5 Design Signal, Not Syntax Sugar

Answer:

1. What is the **only** language-level difference between `struct` and `class`? Structs are private by default, and classes are public
2. Why does C++ even allow both? Because they have different intents
3. When does choosing `struct` communicate intent *better* than `class`? Because being private by default means the data should be protected
4. Why does intent matter more than syntax in large systems? Because intent is easier to interprit

---

Part 5 — Operator Overloading

(Where ADTs Start Feeling Real)

6 Rules You Don't Get to Ignore

Research and explain:

1. Why can't C++ overload:
   - `.` This would break basic functionality of accessing members
   - `::` This is run at compile time
   - `sizeof` not a runtime function
2. Why should `operator+` **not** mutate the left-hand operand? To ensure reliability, and predictability.
3. Why is `operator<<` almost never a member function? Because it must be a non-member to use stream objects

> If your answer is "because that's how everyone does it," dig deeper.

---

7 The "other / rhs" Trap (Yes, It's Intentional)

Given:

```
Point operator+(const Point& rhs) const;
```

Answer clearly:

3. How can this function access `rhs.x` if `x` is private?
4. What does this tell you about **class-level vs object-level access**?

> This question exists specifically to break incorrect mental models.

Part 6 — `friend`: Controlled Violation of Privacy

8 Friend or Design Smell?

Answer:

1. What does the `friend` keyword actually do? Allows a class to access another classes private parts
2. Why is `operator<<` commonly declared as a friend? So it can access components from other classes
3. Why is excessive use of `friend` a red flag? Because some classes could just be public
4. Give **one legitimate use case** and **one illegitimate one**. legit: \ class needs sensitive data from a bad: data should be public

> "Because it wouldn't compile otherwise" is not a justification — it's a symptom.

Part 7 — Core Programming Task

Design a Type, Not a School Assignment

9 Build a Native-Feeling `Point2D`

You are designing a **type**, not checking boxes.

Required Features

Your `Point2D` class must:

- Use **private data members** ✓
- Include at least **three constructors**: ✓
  - default
  - parameterized
  - copy constructor
- Use **initialization lists** ✓
- Overload:
  - `+` ✓
  - `-` ✓
  - `==`
  - `!=` (without duplicating logic) ✓
  - `<<`
- Demonstrate **const correctness** ✓
- Avoid public getters unless you can justify them

```
Point2D a(3, 4);
Point2D b(1, 2);

Point2D c = a + b;
Point2D d = a - b;

if (c == Point2D(4, 6)) {
    std::cout << "Math still works.\n";
}

std::cout << a << std::endl;
```

1️⃣0️⃣ Design Constraints (Read Carefully)

You **may not**:

- expose raw data publicly
- use inheritance
- overload operators with nonsense semantics

You **should**:

- minimize the public interface
- make misuse difficult
- prefer clarity over cleverness

Part 8 — Composition (The Quiet MVP of OOP)

1️⃣1️⃣ Composition in Plain English

> **Composition means assembling behavior from parts, not becoming those parts.**

Or more bluntly:

> "Has-a" beats "is-a" most of the time.

1️⃣2️⃣ Inheritance vs Composition (No Dragons Yet)

❌ Inheritance First Instinct

```
class ColoredPoint : public Point2D {
    Color color;
};
```

Problems:

- Locked into `Point2D` forever
- Inherits everything, wanted or not
- Hard to evolve safely

☑ Composition Approach

```
class ColoredPoint {
private:
    Point2D position;
    Color color;
};
```

Benefits:

- Clear ownership
- Modular behavior
- Fewer unintended side effects

1️⃣3️⃣ The Big Takeaway

> **Inheritance expresses identity.**
> **Composition expresses capability.**

Most real systems care more about **capability**.

Part 9 — Reflection (Yes, This Is Graded)

Answer honestly:

1. Does your `Point2D` *feel* like a built-in type? No
2. What design choice most contributed to that feeling? no other types take input, or i same way
3. Which OOP concept currently feels overhyped? Inheritance
4. Which one feels underrated? encapsulation
5. What part of this worksheet made you uncomfortable — and why? some of the ques were very difficu

If nothing made you uncomfortable, you probably didn't push hard enough.

☑ Grading Rubric (100 Points)

Conceptual Understanding — 25 pts