```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**

That's like saying:

> **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
    - resizing behavior
    - cache friendliness
      (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**?
3. Define the **invariant** for:
    - a stack
    - a queue

> 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	?	?
Requires inheritance	?	?
Same function name	?	?
Same parameter list	?	?

Feature	Overloading	Overriding
Polymorphism involved	?	?

Then answer:

1. Why is **overloading** a *compile-time convenience*?
2. Why is **overriding** a *runtime contract*?
3. Why do beginners confuse the two?
4. Why is that confusion dangerous?

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

Part 3 — Constructors & Initialization Lists

(Where C++ Stops Holding Your Hand)

3️⃣ 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?
2. What happens if you try to assign `id` inside the constructor body?
3. Write the correct constructor.
4. Name **one other situation** where initialization lists are required (research-lite).

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

4️⃣ Copy Constructor vs Assignment Operator

Research + reasoning required.

1. When is the **copy constructor** invoked?
2. When is the **assignment operator** invoked?
3. Why do both exist?
4. What subtle bugs appear if you confuse them?

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

Part 4 — `struct` vs `class`

(Same Machine Code, Different Intent)

5️⃣ Design Signal, Not Syntax Sugar

Answer:

1. What is the **only** language-level difference between `struct` and `class`?
2. Why does C++ even allow both?
3. When does choosing `struct` communicate intent *better* than `class`?
4. Why does intent matter more than syntax in large systems?

Part 5 — Operator Overloading

(Where ADTs Start Feeling Real)

6️⃣ Rules You Don’t Get to Ignore

Research and explain:

1. Why can’t C++ overload:
    - `.`
    - `::`
    - `sizeof`
2. Why should `operator+` **not** mutate the left-hand operand?
3. Why is `operator<<` almost never a member function?

> If your answer is “because that’s how everyone does it,” dig deeper.

7️⃣ The “other / rhs” Trap (Yes, It’s Intentional)

Given:

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

Answer clearly:

1. Which object owns this function?
2. What does `rhs` represent?

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

8️⃣ Friend or Design Smell?

Answer:

1. What does the `friend` keyword actually do?
2. Why is `operator<<` commonly declared as a friend?
3. Why is excessive use of `friend` a red flag?
4. Give **one legitimate use case** and **one illegitimate one**.

> "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

9️⃣ 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

Required Usage (This Must Compile)

```
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?
2. What design choice most contributed to that feeling?
3. Which OOP concept currently feels overhyped?
4. Which one feels underrated?
5. What part of this worksheet made you uncomfortable — and why?

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

✅ Grading Rubric (100 Points)

Conceptual Understanding — 25 pts

- Overloading vs Overriding: 10

- Constructors & Init Lists: 10
- Struct vs Class (intent-based): 5

Operator Overloading Design — 25 pts

- Semantic correctness: 10
- Const correctness: 5
- Member vs friend choice: 5
- Minimal interface: 5

Class Design & Encapsulation — 25 pts

- Private data: 10
- Constructor quality: 5
- Initialization lists: 5
- Friend usage justification: 5

Code Quality — 15 pts

- Compiles cleanly: 5
- Readability & naming: 5
- Comments explain *why*: 5

Reflection & Judgment — 10 pts

- Honest reflection: 5
- Insight into tradeoffs: 5

Final Instructor Note (Read This Twice)

That feeling of:

> "There's always something else… another way… something deeper…"

That's not failure.
That's **abstraction having no ceiling**.

The goal is not perfection.
The goal is **better judgment**.

Deliverables

- Make sure you have a folder called Assignments in your Github repo.
- Create a folder in Assignments called H01 and place all of your documents in there.
- Generate a pdf of your solution and have the entire worksheet not just uploaded to Github, but also printed and brought to class Monday the 16th of February.
- This really is a study guide as we will have an exam the week of the 16th.

- You have to use markdown to create your document. Why? It does syntax highlighting, and makes writing documents easy.
    - Markdown Getting Started
    - Markdown Cheatsheet
- If you use code spaces or download VsCode, you can install plugins to let you convert markdown to pdf, and to allow you to print directly from VsCode.