



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

Stacks & Queues (ADT Reality Check)

Answer **clearly and concisely**.

1. Compare **array-based** vs **list-based** implementations of stacks and queues:

- memory layout
 - An array is stored in memory with all rows occurring consecutively (a sequence)
 - A list's elements can be stored separately anywhere in the memory. Every node points to the next.

- resizing behavior
 - An array cannot be resized once it has been declared
 - A list can be resized after declaration
- cache friendliness
 - Arrays are inherently cache friendly due to the efficiency of their layout in memory (consecutive)
 - Lists are inherently *unfriendly* due to their scattered nature. (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**?

- Stacks "last in first out" works splendidly because it allows access in sequential order without needing to rearrange anything to access the necessary information.

3. Define the **invariant** for:

- a stack
 - Most simply, "last one in, first one out."
- a queue
 - The inverse of the above, "first one in, last one out."
- Simply a set of rules that must remain true throughout the entire process (although they can theoretically be violated in the middle so long as it holds true at the end)

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)

Conceptual Distinction

Fill out the table:

Feature	Overloading	Overriding
Resolved at	Compile	Runtime (Dynamically)
Requires inheritance	No	Yes
Same function name	Yes	Yes
Same parameter list	No	Yes
Polymorphism involved	Yes	Yes

Then answer:

1. Why is **overloading** a *compile-time convenience*?

- The resolution between two methods with different signatures cannot change over time

2. Why is **overriding** a *runtime contract*?

- There is no way to determine the concrete typings or the implementation of certain methods otherwise

3. Why do beginners confuse the two?

- Honestly, I imagine it's primarily the similarity in name. If we're getting more specific, the compiler cannot guess and the programmer must be clear in a way a beginner might not know how to be.

4. Why is that confusion dangerous?

- I wouldn't call it "dangerous," but it can certainly cause issues with Overriding redefining the base class in it's derived class with the same signature.

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

Part 3 — Constructors & Initialization Lists

(Where C++ Stops Holding Your Hand)

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?
 - The usage of the const int **id**
2. What happens if you try to assign **id** inside the constructor body?
 - An error would occur because **id** was already created within said constructor
3. Write the correct constructor.
 - `Widget(const int id, std::string name): ID(id) {};`
4. Name **one other situation** where initialization lists are required (research-lite).
 - If there is not a default constructor.

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

Copy Constructor vs Assignment Operator

Research + reasoning required.

1. When is the **copy constructor** invoked?
 - Creating a default copy constructor, a new object from an existing one

2. When is the **assignment operator** invoked?

- Called when a pre-existing object is assigned a new value from an existing object

3. Why do both exist?

- To assign values to whatever you need them to (in their own specific ways)

4. What subtle bugs appear if you confuse them?

- Memory leaks, overwriting uninitialized memory, and any manner of undefined behavior

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

Part 4 — struct vs class

(Same Machine Code, Different Intent)

Design Signal, Not Syntax Sugar

Answer:

1. What is the **only** language-level difference between struct and class?

- Everything inside a class is private everything inside a struct is public (all by default)

3. Why does C++ even allow both?

- Compatibility with the wider C language

4. When does choosing struct communicate intent *better* than class?

- For the sake of permissions

5. Why does intent matter more than syntax in large systems?

- The purpose behind the code and the choices made thereafter, aka articulating what you want more than anything else. It helps define the purpose more clearly at the human level

Part 5 — Operator Overloading

(Where ADTs Start Feeling Real)

Rules You Don't Get to Ignore

Research and explain:

1. Why can't C++ overload:

- `.`
- `::`
- `sizeof`
- In my personal opinion, it boils down to their closeness to the basics of the language. In theory, you probably could figure out a method to overload these but it would create all sorts of questions and problems when it comes to accessing the members of anything. Pointers would need to be used but they might have ALSO been redefined and could cause more problems.

2. Why should `operator+` **not** mutate the left-hand operand?

- `+` is intended to create new values, not modify old ones

3. Why is `operator<<` almost never a member function?

- It has to do with left side of the operation being devoted to member functions
If your answer is “because that’s how everyone does it,” dig deeper.

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

Given:

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

Answer clearly:

1. Which object owns this function?

- Point

3. What does rhs represent?

- Right-hand side

4. How can this function access rhs.x if x is private?

- Assuming Point operator+ is inside the Point class, then it can simply access the necessary variable

5. What does this tell you about **class-level vs object-level access**?

- Class-level access pertains to the entire class while object-level is contained to the instance of the object. It essentially means object-level instances are protecting object states.

This question exists specifically to break incorrect mental models.

Part 6 — friend: Controlled Violation of Privacy

Friend or Design Smell?

Answer:

1. What does the friend keyword actually do?

- It gives a function or a class access to private members

3. Why is operator<< commonly declared as a friend?

- Giving access to o/istream, aka providing access to these non-extendable libraries

4. Why is excessive use of friend a red flag?

- Because it breaks encapsulation.

5. Give **one legitimate use case** and **one illegitimate one**.

- As mentioned above, using it with `o/iostream` is a proper use while making an entire class a friend could be a problem and, even if not, is bad practice.

“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

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;
```

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)

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.

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

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?

- Yes.

2. What design choice most contributed to that feeling?

- Being able to call `Point2D` like a regular addition/subtraction operator

3. Which OOP concept currently feels overhyped?

- Honestly, I still feel like a child so far. There's so many concepts I've never encountered before. I do not have one- yet.

4. Which one feels underrated?

- The ability to make constructors so concisely! It was awesome getting to use them in this program after seeing the example in class

5. What part of this worksheet made you uncomfortable — and why?

- Honestly? A lot of it. Like I mentioned above, there were so many new and foreign concepts. I'd really love some smaller, more manageable assignments outside of these big

ones to become more familiar with the concepts. It feels like there's a thousand new concepts and I barely recognize them at sight, much less am I capable of implementing them off the top of my head. I know new concepts can take time, so I'm optimistic my opinion will change.

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.