



CC-211L Object Oriented Programming Lab

Department of Computer Science

University of the Punjab

Fall 2026

Lab 5: Composition, Aggregation, and Object Relationships

Lab duration: 1 hour 45 minutes

Part A: Concept check, 10 marks

Part B: Mini-Digger implementation, 90 marks

Grading Scheme

Question:	1	2	3	Total
Points:	10	60	30	100
Score:				

Submission and constraints

- Submit `minidigger.cpp` and one PDF named `roll_number_oop_lab5.pdf` containing the Part A answer and results of both fixed tests.
- Use C++17 and GCC 9.4.
- Use fixed-size arrays. Do not use dynamic memory, STL containers, inheritance, operator overloading, templates, exceptions, or file I/O.
- Keep game state inside the objects. Do not print prompts or output other than the required board and final lines.

Part A: Concept Check

1. [10 marks] A1. Trace references and cascading calls

`Student::setMark` and `Student::setName` each return `Student&`. `StudyPair` stores references to its two constructor arguments and prints:

```
PAIR first=<name>:<mark> second=<name>:<mark>
```

```
Student first{"Areeba", 82};
Student second{"Bilal", 77};
StudyPair pair{first, second};
first.setMark(95).setName("Areeba Khan");
pair.print();
```

Write the exact output. Briefly identify the objects referenced by `pair` and state which objects must outlive it.

Part B: Mini-Digger

Implement a command-driven treasure game on a 6-by-6 grid in `minidigger.cpp`. Coordinates are 1-based. The player starts at row 1, column 1. Commands are **w** (up), **s** (down), **a** (left), and **d** (right). Moving past a grid edge leaves the player in place. Unrecognized commands are ignored.

Required classes and public operations

Implement the five classes with private state. Constructors must initialize all state. Use these operation names and return contracts; private helpers are allowed.

Class	Required state and public operations
Position	Constructor(row, col); store coordinates; four bounded moves return Position& ; const getRow/getCol return int .
Player	Constructor(row, col); own one position and score; four moves return Player& ; mutable/const getPosition ; addScore(int) ; const getScore() returns int .
Treasure	Constructor(row, col, value), initially uncollected; isAt(const Position&) const , isCollected() const , getValue() const , collect() return bool, bool, int, void .
Enemy	Constructor(row, col); isAt(const Position&) const returns bool .
Game	Default constructor; one player, Treasure[4] , Enemy[2] ; display() const , movePlayer(char) , collectTreasure() , hitEnemy() const , finished() const , getScore() const . These return void, four bools, then int .

All data members are private. Position and player movement operations return the receiving object by reference and enforce the grid edges. Position checks borrow the supplied position; treasures and enemies do not store or own it. Use this initial game state:

Object	Row	Column	Value
Player	1	1	—
Treasure 1	1	5	10
Treasure 2	3	3	20
Treasure 3	5	2	30
Treasure 4	6	6	40
Enemy 1	3	6	—
Enemy 2	5	5	—

```
P...$.
.....
..$..E
.....
.$..E.
.....$
```

Collect a matching treasure once only and add its value to the player's score; **collectTreasure** reports whether collection occurred. **movePlayer** returns **true** for a recognized command and **false** otherwise. **finished** is true after an enemy collision or when all treasures are collected. Render six rows of six characters without labels or spaces: P player, E enemy, \$ uncollected treasure, and . empty. For overlaps, use precedence P, E, then \$.

2. [60 marks] B1. Implement Mini-Digger

Implement the classes and behavior above. Marks:

- object state, constructors, composition, and fixed arrays: 15;
- bounded movement and chainable movement interfaces: 10;
- position checks, one-time collection, and scoring: 12;
- collision and completion conditions: 8;
- exact board rendering and required game loop: 15.

3. [30 marks] **B2. Complete and test the program**

In `main`, create one `Game`. With no prompts, repeatedly display the board, read a command, move the player, and check for treasure until the game is finished. Display the final board, then print `GAME OVER` and `SCORE=<score>` on separate lines. Do not create game entities in `main`.

Run each command string as standard input to a fresh program. In the PDF, include the command string and final eight output lines. Each replay is worth 15 marks; the lines and score must match exactly.

Replay 1: collect all treasures

```
ddddssaassasdddd
```

Expected final eight lines:

```
.....  
.....  
.....E  
.....  
.....E.  
.....P  
GAME OVER  
SCORE=100
```

Replay 2: revisit a treasure, test edge moves, then hit an enemy

```
xwadddddssaadadddd
```

Expected final eight lines:

```
.....  
.....  
.....P  
.....  
.$..E.  
.....  
GAME OVER  
SCORE=30
```

Build and submission check

```
g++ -std=c++17 -Wall -Wextra -pedantic minidigger.cpp -o minidigger
```

Submit only `minidigger.cpp` and the PDF. The grader uses the class requirements and fixed replays; no screenshot is required. Do not submit the executable.