**Mostenire**

* ﻿﻿Definiti minim două ierarhii diferite de mostenire (două ierarhii de mos-tenire sunt considerate diferite dacã nu au aceeasi clasã de bazã si aceeasi clasã derivata).

class SpeedyPlayer : public Player {…}

class StealthPlayer : public Player {…}

class Bomb : Weapon {…}

* ﻿﻿Minim o dat membru si minim o metoda care sã aib modificatorul de acces protected (in mod util, sã fie accesate/apelate dintr-o clasă care le mosteneste).

class Player {  
protected:  
 std::string name;  
 bool isAlive;

class Weapon {  
protected:  
 virtual bool isInRange(int targetX, int targetY) = 0;  
};

* ﻿﻿Cel putin o situatie n care sã apelati constructorul (cu parametri) al clasei de bază, folosind lista de initializare din constructorul clasei derivate.

SpeedyPlayer(const std::string& name, int startX, int startY, Map& map, int multiplier = 2)  
 : Player(name, startX, startY, map), speedMultiplier(multiplier) {}

**Metode virtuale si clase abstracte**

* ﻿﻿Definiti si extindeti (mosteniti) minim o clasă abstract (poate avea date membru, dar are cel putin o metoda pur virtuala).

// clasa abstracta cu o metoda pur virtuala

class Weapon {  
protected:  
 virtual bool isInRange(int targetX, int targetY) = 0;  
};

* ﻿﻿Definiti cel putin 2 metode virtuale care s fie suprascrise in clasele mos-tenitoare. Pot fi pur virtuale sau cu o implementare implicită. Se iau în con-siderare si metodele definite la celelalte subpuncte, exceptând destructorii virtuali.

// suprascrierea metodei de mai sus in clasa

class Bomb : Weapon { …

bool isInRange(int targetX, int targetY) override {  
 return abs(targetX - x) <= explosionRange && abs(targetY - y) <= explosionRange;

// Jucatorul se deplaseaza cu deltaX (pe orizontala) si deltaY (pe verticala)  
virtual void move(int deltaX, int deltaY) {  
 int newX = x + deltaX;  
 int newY = y + deltaY;  
 map.updateCell(newX, newY, name[0]); // Actualizăm harta cu noua poziție  
 if(map.getLayout()[y][x]!='x')  
 {map.updateCell(x, y, '.');} // Marchează vechea poziție ca liberă  
 x = newX;  
 y = newY;  
}

// Metoda move din clasa player, suprascrisa in clasa SpeedyPlayer  
void move(int deltaX, int deltaY) override {  
 Player::move(deltaX \* speedMultiplier, deltaY \* speedMultiplier);  
}

**Polimorfism la executie**

* ﻿﻿Identificati si marcati prin câte un comentariu minim 2 instante in care sa aiba loc polimorfism la executie (dynamic dispatch) in proiectul vostru (e.g. apelul unor metode virtuale prin intermediul unor pointeri/referinte catre clasa de bază).

**Sol:**

Polimorfismul la execuție se referă la capacitatea unui program de a alege ce metodă să execute în timpul rulării în funcție de tipul obiectului, nu de tipul referinței sau pointerului. In codul nostru, acest lucru este vizibil prin metodele virtuale ale claselor care sunt suprascrise în clasele derivate.

1. Apelul metodei move() pentru diferite tipuri de jucători (Player, StealthPlayer sau SpeedyPlayer), in clasa Game, in functie de tipul obiectului la runtime:

// Polimorfism la execuție: Apelul se face către move() din Player, StealthPlayer sau SpeedyPlayer, în funcție de tipul obiectului la runtime  
int main() {…

game.getPlayers()[0].move(1, 0); // Player1 se muta la dreapta

2. Apelul metodei isInRange() pentru diferite tipuri de arme, cand bombele sunt explodate în metoda explodeAllBombs() din clasa Game

class Game {…

void explodeAllBombs() {…

if (bomb.isInRange(player.getX(), player.getY())) {  
 // Polimorfism la execuție: Apelul se face către isInRange() din Bomb sau alte clase derivate  
 player.die(); // jucatorii aflati inRange vor muri  
}

﻿﻿Identificati si marcati prin câte un comentariu minim 2 instante de upcasting n codul vostru (e.g. atribuire unor obiecte de tipul unor clase mostenite la pointeri/referinte către clasa de bazã).

**Sol:**

Upcasting este procesul de tratare a unui obiect derivat ca fiind un obiect al clasei de bază. Acest lucru este comun în utilizarea polimorfismului, permițând tratamentul uniform al obiectelor de tipuri derivate.

1. Adaugarea jucatorilor în vectorul de tip Player (upcasting pentru jucatorii de tip StealthPlayer si/sau SpeedyPlayer la clasa de baza Player):

game.addPlayer(player1); // Upcasting: player1 este tratat ca Player, chiar dacă este StealthPlayer sau SpeedyPlayer  
game.addPlayer(player2); // Upcasting: player2 este tratat ca Player, chiar dacă este StealthPlayer sau SpeedyPlayer

2. Accesarea jucătorilor prin referință la clasa de bază în metoda explodeAllBombs()

void explodeAllBombs() { …

for (auto& player : players) {

if (bomb.isInRange(player.getX(), player.getY())) {  
 // Polimorfism la execuție: Apelul se face către isInRange() din Bomb sau alte clase derivate  
 // Upcasting: player este tratat ca referință la clasa de bază Player  
 player.die(); // jucatorii aflati inRange vor muri

**Exceptii**

* ﻿﻿Definiti minim un tip de exceptie custom, care sã extindã clasa exception din biblioteca standard.

#include <exception>  
  
class InvalidMoveException : public std::exception {  
private:  
 std::string message;  
public:  
 InvalidMoveException(const std::string& msg) : message(msg) {}  
 const char\* what() const noexcept override {  
 return message.c\_str();  
 }  
};

* Aruncati exceptii in minim 2 functii/metode diferite (folositi tipul de ex-ceptie definit de voi sau pe cele din biblioteca standard).

void move(int deltaX, int deltaY) {  
 int newX = x + deltaX;  
 int newY = y + deltaY;  
  
 // Verificamd aca noua pozitie este in limitele hartii, altfel aruncam eroare  
 if (newX < 0 || newX > map.getLayout().size() || newY < 0 || newY > map.getLayout()[0].length()) {  
 throw InvalidMoveException("Move out of map bounds!");  
 }

Bomb(int x, int y, int explosionRange)  
 : x(x), y(y), explosionRange(explosionRange) {  
 if (explosionRange < 1) {  
 throw std::invalid\_argument("Explosion range must be at least 1.");  
 }

* Implementati minim un bloc try... catch care s prindã o exceptie definitã si generata de voi (cu specificarea explicit a tipului exceptiei capturate) si sã o trateze într-un fel (sã afiseze un mesaj, sã reîncerce operatiunea, sã arunce o alta exceptie etc).

// Incercam sa incarcam harta din fisier  
try {  
 std::ifstream file("map.txt");  
 if (!file) {  
 throw std::runtime\_error("Unable to open file map.txt - Map file cannot be found or opened.");  
 }  
 game.setMap(file);  
} catch (const std::runtime\_error& e) {  
 std::cerr << "Error: " << e.what() << std::endl;  
 return 1; // intrerupem executia programului daca nu putem incarca harta  
}

**Variabile si metode statice**

* ﻿﻿Definiti si initializati o variabila membru statica in cel putin o clasa.
* ﻿﻿Implementati cel putin două metode statice in clasele voastre (din care cel putin una trebuie sã acceseze/foloseascã variabila static definit la sub-punctul anterior).

class Bomb : Weapon {  
private:  
 static int nr\_bombe;  
 static void atention(){std::cout<<"Atentie, a fost plasata o bomba!!! Total bombe neexplodate: " << std::to\_string(nr\_bombe) << std::endl;}

public:  
 static void decNrOfBombs(){nrOfBombs--; std::cout<<"Atentie, o bomba a explodat! Total bombe neexplodate: " << std::to\_string(nrOfBombs)<< std::endl;}

**Bonus**

* ﻿﻿Separati declaratiile si implementärile din programul vostru folosind fisi-ere header (.h/.hpp) si sursã (.cpp) distincte. Ar trebui sã aveti câte un fisier header si un fisier surs pentru fiecare clas din programul vostru.

﻿﻿Identificati minim o situatie de mostenire în diamant (trebuie sã aib sens pentru tema voastra). Mostenirea în diamant se refer la a avea o clasă de bazã, pe care o mostenese cu virtual două clase distincte, iar apoi aveti o clasã care mosteneste ambele clase intermediare.