**MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**RAPORT**

Lucrare de laborator nr. 3

la cursul ***„Programarea orientată pe obiecte”***

***Tema: Supraîncărcarea operatorilor***

**Varianta 21**

**A efectuat :**   **St. gr. CR-221FR Serba Cristina**

**A verificat: asist. univ. Mantaluța Marius**

**Chișinău 2024**

# Sarcina lucrării:

Să se creeze clasa 2-D de coordonate de tip double în plan. Să se definească operatorii "+" şi "-

" ca funcţii prietene, iar operatorii de atribuire şi de comparare – ca metode ale clasei. De

prevăzut posibilitatea efectuării operaţiilor atît între coordonate, cît şi între coordonate şi

numere obişnuite. Să se definească operatorii "<<" şi ">>"

# Mersul programului:

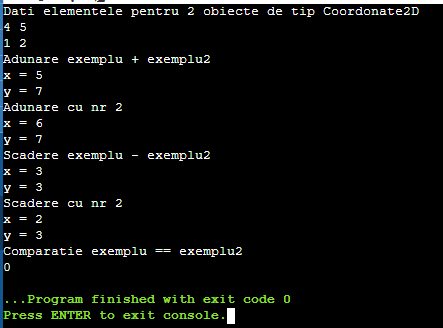
a)

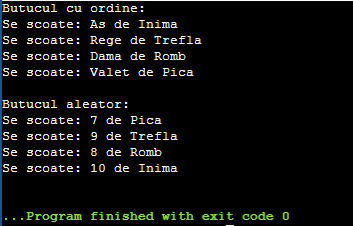
|  |
| --- |
| #include <iostream>  using namespace std;  class Coordonate2D  {  public:  double x;  double y;    Coordonate2D()  {  x = 0;  y = 0;  }    Coordonate2D(double xParam, double yParam)  {  x = xParam;  y = yParam;  }    friend Coordonate2D operator +(const Coordonate2D &first, const Coordonate2D &second);  friend Coordonate2D operator +(const Coordonate2D &example, const double factor);  friend Coordonate2D operator -(const Coordonate2D &first, const Coordonate2D &second);  friend Coordonate2D operator -(const Coordonate2D &example, const double factor);    void operator =(Coordonate2D &example)  {  x = example.x;  y = example.y;  }    bool operator ==(Coordonate2D &second)  {  if ((x == second.x) && (y == second.y))  {  return true;  }  else  {  return false;  }  }    friend ostream& operator <<(ostream &out, Coordonate2D &example);  friend istream& operator >>(istream &in, Coordonate2D &example);  };  Coordonate2D operator +(const Coordonate2D &first, const Coordonate2D &second)  {  Coordonate2D result;  result.x = first.x + second.x;  result.y = first.y + second.y;  return result;  }  Coordonate2D operator -(const Coordonate2D &first, const Coordonate2D &second)  {  Coordonate2D result;  result.x = first.x - second.x;  result.y = first.y - second.y;  return result;  }  Coordonate2D operator +(const Coordonate2D &example, const double factor)  {  Coordonate2D result;  result.x = example.x + factor;  result.y = example.y + factor;  return result;  }  Coordonate2D operator -(const Coordonate2D &example, const double factor)  {  Coordonate2D result;  result.x = example.x - factor;  result.y = example.y - factor;  return result;  }  ostream& operator <<(ostream &out, Coordonate2D &example)  {  out << "x = " << example.x << endl;  out << "y = " << example.y << endl;  return out;  }  istream& operator >>(istream &in, Coordonate2D &example)  {  in >> example.x;  in >> example.y;  return in;  }  int main()  {  Coordonate2D exemplu;  Coordonate2D exemplu2;    cout << "Dati elementele pentru 2 obiecte de tip Coordonate2D\n";  cin >> exemplu;  cin >> exemplu2;    Coordonate2D result = exemplu + exemplu2;  cout << "Adunare exemplu + exemplu2\n" << result;    Coordonate2D result2 = exemplu + 2.0;  cout << "Adunare cu nr 2\n" << result2;    Coordonate2D result3 = exemplu - exemplu2;  cout << "Scadere exemplu - exemplu2\n" << result3;    Coordonate2D result4 = exemplu - 2.0;  cout << "Scadere cu nr 2\n" << result4;    bool resultComparation = exemplu == exemplu2;  cout << "Comparatie exemplu == exemplu2\n" << resultComparation;    return 0;  } |

b)

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <cstdlib>  #include <ctime>  using namespace std;  // Clasa pentru o carte de joc  class CarteDeJoc  {  protected:  string grad;  string culoare;  bool esteDeschisa;  public:  CarteDeJoc()  {  grad = "";  culoare = "";  esteDeschisa = false;  }  CarteDeJoc(const string& gradParam, const string& culoareParam)  {  grad = gradParam;  culoare = culoareParam;  esteDeschisa = false;  }  // Metoda pentru a deschide o carte  void deschide()  {  esteDeschisa = true;  }  // Metoda pentru a intoarce o carte  void intoarce()  {  esteDeschisa = false;  }  // Verifica daca cartea este deschisa  bool esteCarteaDeschisa() const  {  return esteDeschisa;  }  // Supraincarcarea operator afisare  friend ostream& operator <<(ostream& out, const CarteDeJoc& carte)  {  out << carte.grad << " de " << carte.culoare;  return out;  }    // Supraincarcarea operator atribuire  void operator =(const CarteDeJoc &carte)  {  grad = carte.grad;  culoare = carte.culoare;  esteDeschisa = carte.esteDeschisa;  }  };  // Clasa pentru butuc de cărți  class ButucDeCarti {  protected:  vector<CarteDeJoc> carti;  public:  // Adauga o carte in butuc  void adaugaCarte(const CarteDeJoc& carte)  {  carti.push\_back(carte);  }  // Metoda pentru a scoate ultima carte din butuc  virtual CarteDeJoc scoateCarte()  {  CarteDeJoc ultimaCarte;  if(esteGol() == false)  {  ultimaCarte = carti.back();  carti.pop\_back();  }  return ultimaCarte;  }  // Verifica daca butucul este gol  bool esteGol()  {  return carti.empty();  }  };  // Clasa derivata pentru butucul de carti in care cartile trebuie scoase intr-o ordine specifica  class ButucOrdine : public ButucDeCarti  {  private:  int index;  public:  ButucOrdine()  {  index = 0;  }  // Supraincarcarea metodei de scoatere a cartii pentru a fi in ordine  CarteDeJoc scoateCarte() override  {  CarteDeJoc carteOrdine;  if (true == carti.empty())  {  cout << "Butucul de carti este gol!" << endl;  }  else  {  carteOrdine = carti[index];  carti.erase(carti.begin() + index);  if (index > 0)  {  // decrementeaza pana la 0  index--;  }  }  return carteOrdine;  }  };  // Clasa derivata pentru butucul de carti aleatorii  class ButucAleator : public ButucDeCarti  {  public:  // Supraincarcarea metodei de scoatere a cartii pentru a fi aleatorie  CarteDeJoc scoateCarte() override  {  CarteDeJoc carteAleatorie;  if (carti.empty())  {  cout << "Butucul de carti este gol!" << endl;  }  else  {  srand(time(nullptr)); // Initializeaza generatorul de numere aleatoare  int index = rand() % carti.size(); // Genereaza un index aleator  carteAleatorie = carti[index];  carti.erase(carti.begin() + index); // Sterge cartea scoasa din butuc  }  return carteAleatorie;  }  };  int main()  {  // Exemplu de utilizare a claselor pentru butucurile de carti  ButucOrdine butucOrdine;  butucOrdine.adaugaCarte(CarteDeJoc("As", "Inima"));  butucOrdine.adaugaCarte(CarteDeJoc("Rege", "Trefla"));  butucOrdine.adaugaCarte(CarteDeJoc("Dama", "Romb"));  butucOrdine.adaugaCarte(CarteDeJoc("Valet", "Pica"));  cout << "Butucul cu ordine:\n";  while (false == butucOrdine.esteGol())  {  cout << "Se scoate: " << butucOrdine.scoateCarte() << endl;  }  ButucAleator butucAleator;  butucAleator.adaugaCarte(CarteDeJoc("10", "Inima"));  butucAleator.adaugaCarte(CarteDeJoc("9", "Trefla"));  butucAleator.adaugaCarte(CarteDeJoc("8", "Romb"));  butucAleator.adaugaCarte(CarteDeJoc("7", "Pica"));  cout << "\nButucul aleator:\n";  while (false == butucAleator.esteGol())  {  cout << "Se scoate: " << butucAleator.scoateCarte() << endl;  }  return 0;  } |

# Rezultatele obținute:

a) 

b)

# Concluzii:

Moștenirea și compoziția sunt două concepte fundamentale în programarea orientată pe obiecte, care permit dezvoltatorilor să creeze ierarhii de clase și să reutilizeze codul într-un mod eficient. Scopul acestei lucrări a fost de a studia în profunzime aceste două concepte, evidențiind avantajele și dezavantajele fiecăruia, precum și regulile și principiile care le guvernează.

În cadrul lucrării, am analizat moștenirea ca un mecanism prin care o clasă poate să-și extindă funcționalitatea prin preluarea caracteristicilor și comportamentului unei alte clase de bază. Avantajele moștenirii includ reutilizarea codului, extensibilitatea și organizarea logică a codului. Cu toate acestea, trebuie să fim conștienți de dezavantajele moștenirii, cum ar fi complexitatea excesivă, rigiditatea și dependența de structurile de clasă.

În ceea ce privește compoziția, am explorat ideea de a construi obiecte complexe prin combinarea altor obiecte mai simple. Compunerea oferă o flexibilitate mai mare decât moștenirea, deoarece permite modificarea comportamentului la runtime și evită problema ambiguității de la care poate suferi moștenirea.

În concluzie, moștenirea și compoziția sunt două tehnici esențiale în programarea orientată pe obiecte, fiecare având avantaje și dezavantaje specifice. Este important ca dezvoltatorii să înțeleagă în mod clar aceste concepte și să le folosească în mod corespunzător pentru a construi aplicații robuste și ușor de întreținut.