С. А. Вернигоренко
Національний педагогічний університет
імені М.П. Драгоманова

Використання засобів об'єктно-орієнтованого програмування для розвитку пізнавальної активності учнів

Для вчителя, який здійснює творчий підхід у навчанні, завжди існує потреба у підборі цікавих завдань, які б сприяли розвитку пізнавальної та творчої активності його учнів. При цьому, як правило, не виникає особливих труднощів з пошуком умови задачі – математичні теорії дають їх чималу кількість. Тут, зокрема, можуть виникнути наступні запитання:

1. як мотивувати розв'язування;
2. як знайти алгоритм розв'язування;
3. у який спосіб організувати дані у програмі.

Хоча для зацікавлених учнів, у більшості випадків, етап мотивації зводиться до фрази "цікаво, як розв'язати цю задачу?", але все ж бажано, щоб вчитель міг підібрати декілька аргументів, вислухавши які в учня посилилося б бажання здолати поставлену проблему. Особливо це стосується саме задач підвищеної складності, котрі не є обов'язковими і, таким чином, можливість отримати негативну оцінку не слугує для учня вагомим стимулом щодо її розв’язування. Стосовно мотивації розв'язування задач з програмування можна запропонувати наступні аргументи:

1. підібрати задачі з життя, які розв'язуються завдяки безпосередньому використанню поставленої задачі;
2. зауважити учням, що розв'язання даної задачі дозволить у майбутньому розв'язувати ще більш складні завдання, використовуючи дану задачу, як допоміжну;
3. сформулювати умову задачі так, щоб вона сприймалася, як певна проблема з повсякденного життя.

Звичайно, на практиці, можна комбінувати вище описані аргументи, наприклад, спочатку сформулювати задачу у найбільш загальному (абстрактному) вигляді. У якості мотивації використати аргумент 2. Сформулювати більш складну задачу, яка розв'язується з використанням попередньої і для мотивації застосувати третій аргумент.

Стосовно алгоритму розв'язування варто зауважити, що для більшості задач, котрі розглядаються, як задачі певних математичних теорій, у самих цих теоріях описуються алгоритми їх розв'язування. Особливо це стосується задач з теорії графів, мережевого планування, лінійного та цілочисельного програмування і т. д. Тут головна задача вчителя адаптувати ці алгоритми для сприймання учнями. Але поряд з такими задачами можуть виникнути задачі алгоритм розв’язування яких потрібно розробити самостійно, застосовуючи у разі необхідності, знання з певних математичних теорій. Зокрема у цьому полягає елемент творчого підходу у подоланні поставленої проблеми. При розв’язуванні такої задачі, необхідно, щоб вчитель мав у власному розпорядженні принаймні один варіант алгоритму, який він зможе

розтлумачити учням. Хоча може трапитись і так, що учень запропонує більш ефективний алгоритм, ніж той що є у вчителя. Якщо такий факт має місце, то це є однією з підстав вважати, що робота вчителя дає позитивний результат.

Питання щодо способу організації і підбору структур даних у програмі можна вирішувати в залежності від підходу, який застосовується вчителем при навчанні учнів програмуванню. Як при структурному, так і при об'єктно-орієнтованому підході потрібно визначитися з типами даних програми та призначенням підпрограм, які можуть використовуватись у головній програмі. При цьому, якщо дану задачу потрібно буде використати у подальшій роботі, як допоміжну, при розв’язуванні інших задач, то є сенс застосувати саме об’єктно-орієнтований підхід. Аргументацію цього твердження розглянуто нижче на прикладі такої задачі.

Задача. На площині задано множину точок своїми координатами. Координати точок– дійсні числа. Кожна точка має власний номер. Нумерація точок починається з нуля. Серед цих точок знайти номери тих, що утворюють опуклий багатокутник, у якому знаходяться усі інші точки, впорядкувати ці точки (порядок обходу точок повинен задавати опуклий багатокутник). Зрозуміло, що для набору точок, що складається більше ніж з трьох різних точок можна побудувати кілька опуклих багатокутників, але опуклий багатокутник, у якому знаходяться усі точки лише однин. У подальшому називатимемо його "*граничний багатокутник*".

Нижче показано процес розв'язування цієї задачі з використанням мови програмування Сі++.

Початок програми та оголошення класів обробки даних може бути таким:

```
#include <stdio.h>

//клас точки
class CPset2D{
public:
float x, y;
};

//клас множини точок
class CPolyPsets{
public:
//загальна кількість точок у
//множині int n_all;
//кількість точок граничного
//(опуклого) багатокутника
int n_poly;
//вказівник на масив з номерами
//точок граничного багатокутника
int *m_poly;
//вказівник на масив з даними про
// усі точки з множини
CPset2D *m_all;
//порожній конструктор
CPolyPsets(){};
//перевантажені конструктори
CPolyPsets(int N, CPset2D
*M_ALL);
CPolyPsets(FILE *f);
~CPolyPsets();//деструктор
//створення об'єкту
void Create(int N, CPset2D
*M_ALL);
//руйнування об'єкту
void Destroy();

//функція пошуку вершин
// граничного багатокутника
void SerchVertexPolygon();
//функція виведення на екран
//номерів точок граничного
//багатокутника
void PrintNumberPsetPolygon();
private:
//функція обчислення параметрів
//(коефіцієнтів) прямої
//що проходить через точки p1, p2
CPset2D GetParLine(CPset2D p1,
CPset2D p2);
//функція розташування вершин
//граничного багатокутника у
порядку //обходу його контура
void SortVertex();
};
```

Застосування конструкторів класу надає можливість користувачу створювати об'єкт множини точок трьома способами:

1. створення об'єкту без ініціалізації;
2. створення об'єкту з кількістю *N* точок у ньому і заповнення його даними з вказівника на масив *M_ALL*;
3. створення об'єкту і наповнення його даними з файлу.

Крім того у класі окремо розглянуті функції *Create()* та *Destroy(),* які використовуються, відповідно, у конструкторі та деструкторі, а також можуть бути застосовані у головній програмі для динамічного маніпулювання даними об'єкта.

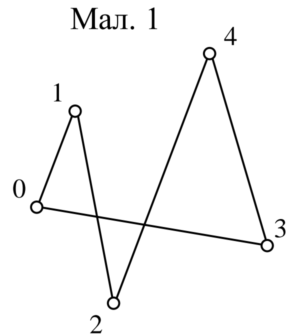
Опис вище згаданих функцій подано нижче.

```
CPolyPsets::CPolyPsets(int N, CPset2D *M_ALL)
{
  Create(N, M_ALL);
}

CPolyPsets::CPolyPsets(FILE *f)
{
  //зчитування даних про
  //координати точок множини з
  //файлу
  fscanf(f, "%d", &n_all);
  n_poly=0;
  m_poly=new int[n_all];
  m_all=new CPset2D[n_all];
  int i=0;
  for(i=0; i<n_all; i++){
      float x, y;
      fscanf(f, "%f%f", &x, &y);
      m_all[i].x=x; m_all[i].y=y;
  }
}

CPolyPsets::~CPolyPsets()
{
  Destroy();
}

void CPolyPsets::Create(int N, CPset2D *M_ALL)
{
  n_all=N;
  n_poly=0;
  m_poly=new int[n_all];
  m_all=new CPset2D[n_all];
  int i;
  for(i=0; i<n_all; i++){
    m_all[i]=M_ALL[i];
    m_poly[i]=-1;
  }
}

void CPolyPsets::Destroy()
{
  delete [] m_poly;
  delete [] m_all;
}
```

Функція *SerchVertexPolygon()* – основна функція класу. Саме у ній реалізований алгоритм пошуку точок, що належать опуклому багатокутнику. Суть алгоритму така. Нехай виконується *k*-тий крок алгоритму (*k=0..n-2, n*– загальна кількість точок у множині). Точка з номером *k* обирається, як базова точка з масиву точок і будується пряма, що з'єднує цю точку з деякою біжучою точкою r *(r=k+1, k+2, ... n-1).* Для усіх точок, крім *k*-тої і *r*-тої, перевіряється їхнє розташування відносно побудованої прямої. Якщо існують точки, розташовані по різні боки від прямої, то точки з номерами *k* і *r* не є вершинами шуканого багатокутника, інакше обидві ці точки фіксуються, як вершини багатокутника. Після перебору усіх можливих прямих, що проходять через базову *k*-ту точку та біжучу *r*-ту точку за базову обирають *k+1*-шу точку і весь процес повторюють знову. Пошук продовжується до тих пір, поки базовою не стане точка з номером *n-1* (остання точка з множини точок).

У деяких випадках застосування вище описаного алгоритму надає можливість отримати номери точок багатокутника у порядку обходу його контура. Але можливі ситуації, коли знайдені номери вершин не відповідають порядку обходу контура (див. мал. 1). Для ситуації на малюнку, відповідно до алгоритму, отримається такий порядок обходу вершин: 0, 1, 2, 4, 3, 0. Тому виникає потреба створити допоміжну функцію, застосування якої надавало б можливість розташовувати номери точок у правильному порядку.

Мал. 1



Алгоритм цієї функції може бути таким. Перевірити, чи існує точка перетину *k*-тої сторони багатокутника з *k*+2-ою, *k*+3-тьою, і т. д. (*k*=1, 2, 3, …, *n*, де *n* – кількість сторін багатокутника). Якщо для деякої пари сторін існує точка перетину, то у наборі чисел, який визначає порядок обходу контура багатокутника поміняти місцями номер другої вершини *k*-тої сторони і номер першої вершини *k+j*-тої сторони. Процес продовжується доти, доки існують точки перетину сторін. Наприклад, з малюнка видно, що сторона (1, 2) перетинається стороною (3, 0). Отже у числовому наборі, в якому вказується порядок обходу контура, потрібно поміняти місцями числа 2 і 3. Отримається набір 0, 1, 3, 4, 2, 0. Далі очевидно, що сторони (1, 3) і (4, 2) мають спільну точку, отже, відповідно до алгоритму потрібно поміняти місцями числа 3 і 4. Одержиться послідовність 0, 1, 4, 3, 2, 0, котра визначає порядок обходу вершин багатокутника.

Такий алгоритм впорядкування точок багатокутника реалізований у класі множини точок у вигляді функції *SortVertex()*, яка використовується в кінці функції *SerchVertexPolygon()*.

Опис функцій *SerchVertexPolygon()* та *SortVertex()* показаний нижче.

Функція SerchVertexPolygon()

```
void
CPolyPsets::SerchVertexPolygon()
{
  int i;
  //цикл перебору точок
  for(i=0; i<n_all-1; i++){
    int j=i+1;
    /*цикл перебору прямих, що
    проходять через i-ту точку та
    усі інші точки, номери яких
    більші від номеру i-тої точки*/
    while (j<n_all){
//обчислення параметрів //прямої
CPset2D par_line;
            par_line=GetP
            arLine(m_all[i
            ], m_all[j]);
//прапор розташування точок //у
одній півплощині
bool one_side=true;
//лічильник точок,
//розташованих у першій
//півплощині і на прямій
int side1=0;
```

```
      //лічильник точок,
      //розташованих у другій
      //півплощині
      int side2=0;
      int k=-1;
      /*
      цикл визначення взаємного
      розташування прямої, заданої
      параметрами par_line і усіх
      точок множини
      */
  while (++k<n_all && one_side){
    //якщо k-та точка не
    //співпадає з точками прямої
    if (k!=i && k!=j){
      float mean;
      //якщо пряма
      //не паралельна осі ОУ
      if (par_line.x!=1000 ||
      par_line.y!=1000){
        mean=m_all[k].y-
        par_line.x*m_all[k].x-
        par_line.y;
        if (mean>0)
          side1++;
        else
          side2++;
    }
    else{//якщо пряма паралельна
          //осі ОУ
      if (m_all[k].x<m_all[i].x)
        side1++;
      else
          side2++;
    }
  }
  if (side1>0 && side2>0)
    one_side=false;
}
//додавання номерів точок до
масиву //номерів вершин опуклого
//багатокутника
if (one_side){
    /* перевірка належності і-тої та
    j-тої точки до масиву номерів
    вершин багатокутника */
    int ii=-1;
    //прапор додавання і-тої точки
    //до масиву
    bool add_i=true;
    //прапор додавання j-тої точки
    //до масиву
    bool add_j=true;
    while (++ii<n_poly){
      if (m_poly[ii]==i)
          add_i=false;
      if (m_poly[ii]==j)
          add_j=false;
    }
    if (add_i)
         m_poly[n_poly++]=i;
    if (add_j)
         m_poly[n_poly++]=j;
      }
      j++;
    }
  }
  //сортування точок багатокутника
  //у порядку обходу контура
  SortVertex();
}
```

Функція SortVertex()

```
void CPolyPsets::SortVertex()
{
  int i=0, j;
  bool b_cross;
  do{
    b_cross=false;
    for(i=0; i<n_poly; i++){
      int i_a, i_b;
      i_a=m_poly[i];
      i_b=m_poly[(i+1)%n_poly];
      int swap_i=(i+1)%n_poly;
      CPset2D A(m_all[i_a].x,
      m_all[i_a].y);
      CPset2D B(m_all[i_b].x,
      m_all[i_b].y);
      CLine2D ab(A, B);
      j=0;
      while(j<n_poly-i-2 && !b_cross){
        int i_c, i_d;
        i_c=m_poly[i+j+2];
        i_d=m_poly[(i+j+3)%n_poly];
        int swap_j=i+j+2;
        CPset2D C(m_all[i_c].x,
        m_all[i_c].y);
                  CPset2D
                  D(m_all[i_d].x,
                  m_all[i_d].y);
        CLine2D cd(C, D);
        CPset2D cross;
        cross=ab*cd;
        if (cross.x==-1 && cross.y==-1 ||
        cross.x==A.x && cross.y==A.y ||
```

```
cross.x==B.x &&
cross.y==B.y ||
cross.x==C.x &&
cross.y==C.y ||
cross.x==D.x &&
cross.y==D.y){
  ;
}
else{
        m_poly[swap_i]=m_poly[swap
        _j];
        m_poly[swap_j]=i_b;
        b_cross=true;
      }
      j++;
    }
  }
}while (b_cross);
}
```

Як видно з опису функції *SortVertex(),* у її тілі використовується клас *CLine2D*, що моделює відрізок. Крім цього далі у цій статті буде використовуватись клас трикутника. Такі класи учні можуть створити раніше – на початку знайомства з елементами об’єктно-орієнтованого програмування. Причому можна запропонувати наступну послідовність їх створення. На початку ознайомлення з темами, які пов’язані з поняттями класу та об’єкта, учні створюють класи точки та відрізка, як окремі класи. При вивченні поняття успадкування класів вони мають встановити зв’язок між ними, а також породити від класа відрізка клас трикутника. Таким чином, розгляд вище зазначених понять є пропедевтикою до розв’язування більш складних задач. У подальшому, при вивченні питань, пов’язаних з перевантаженням стандартних операцій, корисно розібрати з учнями вправу на перевантаження операції множення для об’єктів відрізків. Результатом виконання такої операції є об’єкт точки, поля якого зберігають координати точки перетину відрізків над якими виконувалась операція множення, при умові, що відрізки мають спільну точку. Операція множення відрізків застосовується в тілі функції *SortVertex()* для визначення точки перетину двох сторін багатокутника. Текст функції перевантаження операції множення для об’єктів відрізків подано нижче.

```
CPset2D
operator*(CLine
2D &line)
{
  float a[2][3];
  a[0][0]=x2-x;
  a[0][1]=line.x-line.x2;
  a[0][2]=line.x-x;

  a[1][0]=y2-y;
  a[1][1]=line.y-line.y2;
  a[1][2]=line.y-y;

  float d=a[0][0]*a[1][1]-
  a[0][1]*a[1][0];
if (d==0)
  return CPset2D(-1.1, -1.1);
else{
  float t1, t2;
  t1=(a[0][2]*a[1][1]-
  a[0][1]*a[1][2])/d;
  t2=(a[0][0]*a[1][2]-
  a[0][2]*a[1][0])/d;
      if (t1>=0 && t1<=1 && t2>=0
      && t2<=1)
              return
              CPset2D(a[0][0]*
              t1+x,
              a[1][0]*t1+y);
  else
```

```
        return CPset2D(-1.1, -1.1);            }
  }
```

В описі цієї функції враховані усі можливі "критичні випадки" такі, як розташування відрізків на паралельних прямих, співпадання відрізків або їх частин, існування точки перетину прямих які містять відрізки, але не існування точки перетину самих відрізків. Для таких випадків результатом виконання операції множення над об’єктами двох відрізків є точка з координатами (-1.1; -1.1). При написанні функції у якості моделі прямої було обрано параметричне подання прямої. Матеріал, що стосується параметричного рівняння прямої не входить до шкільної програми з математики, але він є не занадто складним і тому вчитель може повідомити учням суть параметричного подання прямої, що проходить через дві точки і розтлумачити звідки отримуються оператори описані в функції перевантаження операції множення.

Функція *PrintNumberPsetPolygon()* призначена для виведення на екран у консольному режимі номерів точок багатокутника.

Крім вище згаданих функцій, у класі множини точок є допоміжна функція *GetParLine(),* яка призначена для отримання коефіцієнтів прямої що проходить через вказані точки. Оскільки ця функція допоміжна, то саме з цієї причини її оголошення у класі можна розташувати у приватній секції, як це і зроблено у вище наведеному оголошенні класу множини точок.

Опис функцій *PrintNumberPsetPolygon()*, *GetParLine()* подано нижче.

```
void CPolyPsets::PrintNumberPsetPolygon()
{
    int i;
    for (i=0; i<n_poly; i++)
        printf("%d ", m_poly[i]);
    printf("\n");
}
                                        CPset2D CPolyPsets::GetParLine(
                              CPset2D p1, CPset2D p2)
{
  CPset2D data;

  //визначення коефіцієнтів k і b прямої y=k*x+b
  if (p2.x-p1.x!=0){
    data.x=(p2.y-p1.y)/(p2.x-p1.x);//k
    data.y=(p1.y*p2.x-p1.x*p2.y)/(p2.x-p1.x);//b
  }
  else{
    data.x=1000;//k
    data.y=1000;//b
  }

  return data;
```

```
}
```

Далі описано дві можливі реалізації головної функції програми. У одній з них спочатку створюється і наповнюється даними динамічний масив точок, після чого такий масив передається конструктору об’єкта множини точок; у іншій – об’єкт множини точок створюється конструктором, що дозволяє наповнити його даними з текстового файлу. Формат подання даних у файлі досить простий: спочатку вказується кількість точок множини, потім, з використанням символів пробілу та кінця рядка, пари координат цих точок.

```
void main(void)
{
  CPset2D *data;
  int n;
  printf("Input n=");
  scanf("%d", &n);
  data=new CPset2D[n];
  int i;
  for(i=0; i<n; i++){
    printf("Input p[%d].x=", i);
    scanf("%f", &data[i].x);
    printf("Input p[%d].y=", i);
    scanf("%f", &data[i].y);
  }
  CPolyPsets ps(4, data);
ps.SerchVertexPolygon();
ps.PrintNumberPsetPolygon();
delete [] data;
}

void main(void)
{
    FILE *f;
    f=fopen("data2.txt", "r");
    CPolyPsets ps2(f);
    fclose(f);
    ps2.SerchVertexPolygon();
    ps2.PrintNumberPsetPolygon();
}
```

Бажано звернути увагу учнів на той факт, що отриманий клас множини точок можна використати, як клас опуклого багатокутника і функцію *SerchVertexPolygon()* застосовувати для розташування номерів точок багатокутника у порядку обходу його контура.

Варто зауважити, що для розв’язування учнями поставленої задачі з використанням ООП, необхідно щоб у них були сформовані уявлення про наступні поняття: "клас", "об’єкт", "секції оголошення класу", "опис функцій класу", "конструктор", "деструктор" (див. [1]). Крім того важливо, щоб учні чітко розуміли механізми виклику конструкторів та деструктора класу. Це особливо актуально при роботі з динамічними об’єктами класу, оскільки у такому випадку задача виділення-звільнення пам’яті під об’єкти повністю покладається на програміста.

Маючи у своєму розпорядженні таку задачу і розглянувши процес її розв’язування з учнями, вчитель може запропонувати їм наступні вправи:

1. Додати у клас множини точок функцію *InputData()* для введення з клавіатури даних про цю множину (кількість точок та координати точок).

2. Змінити текст головної програми так, щоб об’єкт множини точок створювався з використанням порожнього конструктора і наповнювався даними із застосуванням функції *InputData()*.

3. Додати у клас множини точок функцію *GetLength()* для обчислення периметру граничного багатокутника.

4. Додати у клас множини точок функцію *GetSquare()* для обчислення площі граничного багатокутника.

5. Створити динамічний масив множин точок. Наповнити його даними. Серед елементів масиву знайти багатокутник з найменшим (найбільшим) периметром та найменшою (найбільшою) площею.

При розв’язуванні цих вправ потрібно врахувати, що оскільки у функціях для підрахунку периметра і площі багатокутника доводиться оперувати номерами вершин багатокутника, то необхідно попередньо перевірити чи не порожній динамічний масив, що містить такі номери і у випадку, коли він порожній викликати функцію *SerchVertexPolygon()*.

Нижче подано опис функцій підрахунку периметру та площі.

```
float CPolyPsets::GetLength()
{ float P=0;
    int i, j, k;
    if (m_poly[0]<0)
        SerchVertexPolygon();
    for(i=0; i<n_poly; i++){
        j=m_poly[i];
        k=m_poly[(i+1)%n_poly];
        CLine2D          line(m_all[j],
        m_all[k]);
        P+=line.GetLength();
    }
    return P;
}

float CPolyPsets::GetSquare()
{
    float S=0;
    int i, j, k;
    if (m_poly[0]<0)
        SerchVertexPolygon();
    for(i=0; i<n_poly-2; i++){
        j=m_poly[i+1];
        k=m_poly[i+2];
                CTriangle2D
                triangle(m_all[0],
                m_all[j], m_all[k]);
    S+=triangle.GetSquare();
    }
    return S;
}
```

При подальшому вивченні основ ООП, а саме при знайомстві з питаннями, пов’язаними з перевантаженням стандартних операцій, можна запропонувати учням таку задачу: з літака на ворожу територію було скинуто два десантних батальйони. При посадці кожен боєць кожного батальйону передав на командний пункт власні координати. Визначити, яку максимальну площу покрили десантні батальйони.

Фактично розв’язування задачі зводиться до побудови двох об’єктів множин точок і побудови третього об’єкта, який є об’єднанням двох попередніх. Отримавши третій об’єкт і обчисливши його площу матимемо відповідь на питання задачі. Операцію об’єднання можна позначити символом "+" і перевантажити її за таким правилом "об’єднанням двох множин

точок є третя множина, яка складається з усіх точок, що належать першій або другій множині". Очевидно це правило є частинним випадком означення об'єднання елементів двох множин довільної природи, що формулюється в теорії множин. Крім цього для коректної роботи з виразами, що містять операцію додавання, необхідно перевантажити операцію присвоєння та конструктор копіювання.

Далі показано процес оголошення та опису перевантажених операцій присвоєння, додавання та конструктора копіювання.

Фрагмент оголошення класу

```
…      …      …
//конструктор копіювання
CPolyPsets(CPolyPsets &ps);
…      …      …
//перевантаження операції присвоєння
CPolyPsets & operator=(CPolyPsets &a);
//перевантаження операції додавання
CPolyPsets operator+(CPolyPsets a);
…      …      …
```

Опис перевантажених функцій

```
CPolyPsets & CPolyPsets::operator = (CPolyPsets &a)
{
   if (n_all>0) Destroy();
   Create(a.n_all, a.m_all);
   SerchVertexPolygon();
   return *this;
}
```

```
CPolyPsets CPolyPsets::operator +(CPolyPsets a)
{
   CPset2D *psets;
   int n=n_all+a.n_all;
   psets=new CPset2D[n];
   int i, j;
   for(i=0; i<n_all; i++)
         psets[i]=m_all[i];
   for(j=0; j<a.n_all; j++)
         psets[j+i]=a.m_all[j];
   CPolyPsets b;
   b.Create(n, psets);
   delete [] psets;

   return CPolyPsets(b);
      }
```

На прикладі перевантаження операцій можна закріпити знання учнів про поняття "конструктор копіювання" і "конструктор перетворення" та послідовність їх виклику.

Важливо зауважити, що вище поданий опис функції, використання якої реалізує перевантаження операції додавання не враховує ситуації, коли точки двох множин мають однакові координати. Тому можна запропонувати учням змінити код перевантаження операції додавання так, щоб не відбувалося дублювання даних для точок з однаковими координатами.

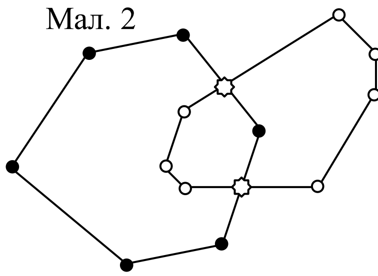
Фрагмент головної програми, у якому відбуватиметься додавання об'єктів та підрахунок площі отриманого об'єкту матиме, зокрема, такий вигляд.

```
…      …      …
FILE *f;

f=fopen("data2.txt", "r");
CPolyPsets ps2(f);
fclose(f);

f=fopen("data3.txt", "r");
CPolyPsets ps3(f);
fclose(f);
```

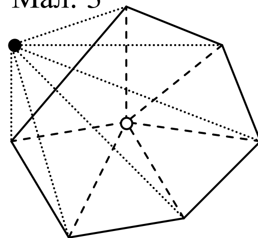
```
CPolyPsets ps4;                    printf("S=%.3f\n",
                                   ps4.GetSquare());
ps4=ps2+ps3;                       …       …       …
```

Можна також запропонувати учням задачу, що є модифікацією попередньої: визначити, який з батальйонів при посадці покрив більшу площу.

Тут корисно підказати учням здійснити перевантаження операцій порівняння ">", "<", "==", ">=", "<=", "!=". При цьому за критерій порівняння можна обрати таке твердження "з двох множин точок "більшою" є та, у якої площа граничного багатокутника більша".


Мал. 2

Крім вище описаних задач варто розглянути ще й таку: два опуклі багатокутники задані координатами своїх вершин, причому вершини можуть вказуватись у довільному порядку (не тільки в порядку обходу контура багатокутника). Обчислити площу спільної частини цих багатокутників (див. мал. 2). Алгоритм розв'язування даної задачі може бути наступним:


Мал. 3

1. Знайти точки першого багатокутника, які знаходяться у середині другого.
2. Знайти точки другого багатокутника, які знаходяться у середині першого.
3. Знайти точки перетину сторін багатокутників.
4. Із усіх знайдених точок побудувати новий багатокутник та обчислити його площу.

Можна розв’язати цю задачу використовуючи раніше створений клас *CPolyPsets*. Для цього, зокрема, потрібно у секцію приватних оголошень додати оголошення допоміжної функції *bool PsetIn(CPset2D pset)* використання якої надасть можливість визначати належність точки *pset* до граничного багатокутника множини точок. У найпростішому випадку перевірку належності точки опуклому багатокутнику можна здійснити порівнявши площу багатокутника з сумою площ трикутників однією з вершин яких є точка *pset*, а інші дві вершини – сусідні вершини багатокутника (див. мал. 3). Іншими словами потрібно порівняти площу заданого багатокутника з площею багатокутника, що отримується з множини точок заданого плюс точка *pset*. Саме такий підхід реалізовано в нижче описаній функції *PsetIn()*.

```
bool CPolyPsets::PsetIn(CPset2D pset)
{ CPset2D *data;
  data=new CPset2D[n_all+1];
  for(int i=0; i<n_all; i++) data[i]=m_all[i];
  data[i]=pset;
  CPolyPsets ps(n_all+1, data);
  delete [] data;
  if (ps.GetSquare()<=GetSquare()) return true;
  else return false;
}
```

маючи функцію *PsetIn()* та функцію перевантаження операції множення для об'єктів відрізків (опис такої функції розглянуто вище) не складно створити функцію для побудови нового багатокутника, який є перетином двох багатокутників (далі називатимемо його "*результуючий багатокутник*"), при умові, що такий перетин існує. Цю функцію можна описати, як перевантаження операції множення для об'єктів багатокутників. Одна з можливих реалізацій такого перевантаження описана нижче.

```
CPolyPsets
CPolyPsets::operator
* (CPolyPsets &a)
{
   if (n_poly<=0) SerchVertexPolygon();
   if (a.n_poly<=0)
     a.SerchVertexPolygon();
   //вказівник на масив точок
   //результуючого багатокутника (РБ)
   CPset2D *psets;
   //точка перетину сторін
   //багатокутників
   CPset2D cross;
   //максимально можлива
   //кількість точок РБ
   int max;
     if (n_poly>a.n_poly) max=n_poly;
   else
     max=a.n_poly;
   psets=new CPset2D[max];
   //кількість точок РБ
   int b_n_poly=0;
   //кількість точок першого
   //багатокутника, які належать
   //другому
   int p_InL=0;
   //кількість точок другого
   //багатокутника, які належать
   //першому
   int p_InR=0;
   //пошук усіх точок першого
   //багатокутника, які належать
   //другому
   int i;
   for(i=0; i<n_poly; i++)
  if (a.PsetIn(m_all[m_poly[i]])){
  if (!SamePset(b_n_poly,
                    psets,
                    m_all[m_poly
                    [i]])){
            p_InL++;
            psets[b_n_poly++]=m_all[
            m_poly[i]];
          }
       }
   //пошук усіх точок другого
   багатокутника,
   //які належать першому
   for(i=0; i<a.n_poly; i++)
        if
        (PsetIn(a.m_all[a.m_poly[i]]))
        {
  if (!SamePset(b_n_poly,
                    psets,
                    a.m_all[m_pol
                    y[i]])){
            p_InR++;
            psets[b_n_poly++]=a.m_al
            l[a.m_poly[i]];
          }
       }
```

```
//пошук точок перетину //сторін
багатокутників
int j;
for(i=0; i<n_poly; i++){
  CLine2D ll(m_all[m_poly[i]],
m_all[m_poly[(i+1)%n_poly]]);
for(j=0; j<a.n_poly; j++){
    CLine2D lr(a.m_all[a.m_poly[j]],
    a.m_all[a.m_poly[(j+1)%a.n_poly]])
    ;
    cross=ll*lr;
    if (cross.x!=-1 && cross.y!=-1){
        if (!SamePset(b_n_poly,
        psets, cross))
          psets[b_n_poly++]=cross;
      }
    }
  }
  //об'єкт РБ
  CPolyPsets b(b_n_poly, psets);
  delete [] psets;
  b.SerchVertexPolygon();
  return CPolyPsets(b);
}
```

В описі цієї функції, крім функції *PsetIn()*, використано допоміжну функцію *SamePset()*, яка застосовується для встановлення факту існування у деякому масиві точок точки з вказаними координатами. Використання цієї функції надає можливість не записувати до масиву точок результуючого багатокутника точки з однаковими координатами. Ситуація при якій може трапитися, що у масиві з'являться дві точки з однаковими координатами має місце у тому випадку, коли два багатокутники мають спільні вершини або їх сторони, частково чи повністю, співпадають.

Далі наводиться одна з можливих реалізацій функції *SamePset()*.

```
bool CPolyPsets::SamePset(int n, CPset2D m[], CPset2D p)
{
    //тут n – кількість точок у масиві m,
    //m – масив, що містить об'єкти точок,
    //p – точка, для якої здійснюється перевірка її належності до масиву m
    bool same_pset=false;
    for(int j=0; j<n && !same_pset; j++)
        if (m[j]==p) same_pset=true;
    return same_pset;
}
```

У тексті функції зустрічається вираз "m[j]==p" – перевірка існування точки p у масиві m. Фактично тут відбувається порівняння двох об'єктів класу "точка" і тому необхідно перевантажити операцію рівності для класу "точка". Очевидно, що правило, за яким здійснюватиметься перевантаження, має бути таким: "дві точки вважатимемо "рівними", якщо їх відповідні координати однакові".

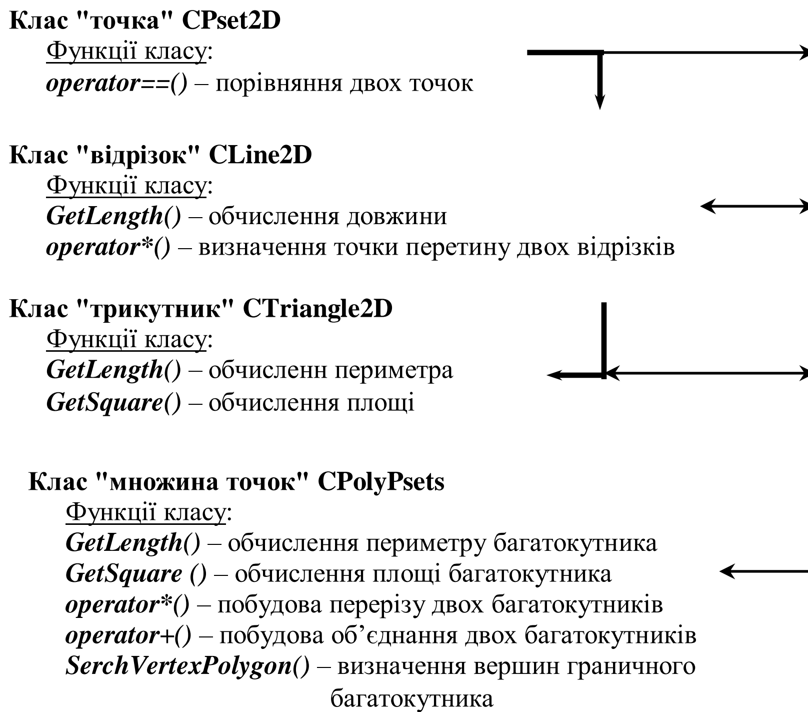
Таким чином, після опису всіх допоміжних функцій, фрагмент головної програми, у якому здійснюється обчислення спільної площі двох багатокутників може бути таким.

```
…        …        …                    //сворення першого
                                       //багатокутника
```

```
f=fopen("poly1.txt", "r");
CPolyPsets poly1(f, 2);
fclose(f);
//створення другого
//багатокутника
f=fopen("poly2.txt", "r");
CPolyPsets poly2(f, 2);
fclose(f);

//опис об'єкту результуючого
//багатокутника
CPolyPsets poly;
//побудова об'єкту
//результуючого багатокутника
poly=poly1*poly2;
//обчислення площі
//результуючого багатокутника
printf("S=%.3f\n",
poly.GetSquare());
…          …          …
```

Після розгляду усіх вище описаних задач, вчителю разом з учнями доцільно зробити підсумок виконаної роботи. Результатом підсумку може бути схема класів на якій показані назви класів та коротка анотація їх функцій (див. мал. 4).

мал. 4



На схемі напівжирними стрілками позначені зв'язки успадкування класів, тонкими стрілками – використання одних класів іншими, тобто агрегація класів.

На базі вище розглянутих задач можна ставити перед учнями нові задачі причому для їх розв'язання не доведеться створювати все з початку. Так, наприклад, до класу "відрізок" можна додати функцію для визначення взаємного розташування деякої точки і відрізка. До класу "трикутник" можна додати функції для обчислення елементів трикутника: основ бісектрис, медіан, висот та їх довжини; центри вписаного та описаного кіл та їх радіуси і т. д. (див. [2], [3]). В подальшому вчителю бажано дібрати задачі, розв'язування яких потребує застосування вище перерахованих функцій.

Головна складність використання об'єктно-орієнтованого підходу при розв'язуванні задач полягає в тому, що учні повинні володіти певними знаннями з теорії ООП. Але, разом з цим, застосування такого підходу має виробляти у них вміння розбивати задачу на підзадачі, встановлювати зв'язки між окремими компонентами задачі та ефективно використовувати ці зв'язки. На думку автора саме у такому аспекті ООП може слугувати альтернативою структурованому програмуванню.

ЛІТЕРАТУРА

1. Майкл Дж. Янг VisualC++ 6. Полное руководство: Пер. с англ. – К.: Издательская группа BHV, 2000. – 1056 с., ил.
2. http://algolist.manual.ru/maths/geom/datastruct.php
3. http://alglib.manual.ru/geometry/