Пенко В., Пенко О. Використання візуалізації на різних етапах вивчення дисципліни «Програмування». *Освіта. Інноватика. Практика,* 2023. Том 11, № 2. С. 31-39. DOI: 10.31110/2616-650X-vol11i2-005

Penko V., Penko O. Vykorystannia vizualizatsii na riznykh etapakh vyvchennia dystsypliny «Prohramuvannia» [Using visualization at different stages of studying the discipline "Programming"]. *Osvita. Innovatyka. Praktyka – Education. Innovation. Practice,* 2023. Vol. 11, No 2. S. 31-39. DOI: 10.31110/2616-650X-vol11i2-005

УДК 004.42
DOI: 10.31110/2616-650X-vol11i2-005

Валерій ПЕНКО
Одеський національний університет імені І.І. Мечникова, Україна
https://orcid.org/0000-0002-0190-6694
vpenko@onu.edu.ua
Олена ПЕНКО
Комунальний заклад «Рішельєвський науковий ліцей», Україна
https://orcid.org/0000-0002-1488-3532
elenapenko@onu.edu.ua

ВИКОРИСТАННЯ ВІЗУАЛІЗАЦІЇ НА РІЗНИХ ЕТАПАХ ВИВЧЕННЯ ДИСЦИПЛІНИ «ПРОГРАМУВАННЯ»

Анотація. *У статті розглядається синергетичний процес використання засобів візуалізації та інструментів мови програмування для вирішення декількох типів завдань програмування. Така синергія виникає у зв'язку з орієнтованістю когнітивних функцій мозку на обробку саме візуальної інформації. Для реалізації сформульованих цілей авторами був описаний процес поетапного вирішення декількох учбових завдань з використанням мови програмування C# та відповідними графічними об'єктами. Таке подання процесу вирішення задач сприяє ефективному обранню адекватних програмних конструкцій та засобів. Розглянуті завдання пов'язані з двовимірним простором, що моделюється в програмування двовимірними масивами. З точки зору авторів саме двовимірні масиви є найбільш поширеною інформаційною структурою, яка дозволяє використовувати візуальне представлення як ефективний спосіб пошуку та пояснення процесу вирішення. Найчастіше в такому випадку використовуються вкладені цикли, що обстежують вміст масиву природним способом. До того ж розглянуто приклад задачі, де вкладені цикли не можуть бути застосовані. Запропоновано декілька доречних прийомів, що спрощують сприйняття програмного коду, а саме використання функцій (методів), перерахованих типів та заміна складної логіки в умовних операторах на масиви, що кодують елементи умов. Завдання, що розглянуто: визначення площі трикутника шляхом підрахунку кількості його клітинок, виявлення результати гри у хрестики-нулики та заповнення квадратного масиву послідовністю цілих чисел по спіралі. Вирішення кожного завдання супроводжується основним програмним кодом. Для кращого засвоєння викладеного матеріалу, учні мають сконструювати повний робочий код та виконати самостійні завдання, що містяться у тексті статті.*

Ключові слова: *програмування; інформатика; візуалізація; ігрове поле; хрестики-нулики; масиви; квадратна матриця; C#; заповнення по спіралі; алгоритм.*

Valerii PENKO
Odessa I. I. Mechnikov National University, Ukraine
https://orcid.org/0000-0002-0190-6694
vpenko@onu.edu.ua
Olena PENKO
Communal institution "Richelieu Scientific Lyceum", Odessa, Ukraine
https://orcid.org/0000-0002-1488-3532
elenapenko@onu.edu.ua

USING VISUALIZATION AT DIFFERENT STAGES OF STUDYING THE DISCIPLINE "PROGRAMMING"

Abstract. *The article examines the synergistic process of using visualization tools and programming language tools to solve several types of programming tasks. Such a synergy arises in connection with the orientation of the cognitive functions of the brain to the processing of visual information. In order to realize the stated goals, the authors described the process of step-by-step solving of several educational tasks using the C# programming language and corresponding graphic objects. Such presentation of the problem-solving process contributes to the effective selection of adequate software structures and tools. The considered tasks are related to two-dimensional space modeled in programming by two-dimensional arrays. From the point of view of the authors, it is two-dimensional arrays that are the most common information structure that allows using visual representation as an effective way of searching and explaining the solution process. Most often, in this case, nested loops are used, which examine the contents of the array in a natural way. In addition, an example of a problem where nested loops cannot be applied is considered. A number of appropriate techniques are proposed that simplify the perception of the program code, namely the use of functions (methods), enumerated types, and the replacement of complex logic in conditional operators with arrays encoding elements of conditions. The task considered: determining the area of a triangle by counting the number of its cells, finding the results of the tic-tac-toe game, and filling a square array with a sequence of integers in a spiral. The solution of each task is accompanied by the main program code. For better assimilation of the presented material, students should construct a complete working code and complete the independent tasks contained in the text of the article.*

Keywords: *programming; Computer Science; visualization; playing field; tic-tac-toe; arrays; square matrix; C#; spiral filling; algorithm.*

Однією з найефективніших технологій активізації навчання є метод візуалізації навчальної інформації, який міцно зайняв своє місце в освітньому процесі. Застосування візуальних форм засвоєння навчальної інформації дозволяє змінити характер навчання: прискорити сприйняття, осмислення та узагальнення, уміння аналізувати поняття, структурувати інформацію.

Методичний інтерес представляє використання візуалізації на різних етапах вивчення навчальної дисципліни. Це твердження ґрунтується на досить добре підтверджених експериментальних дослідженнях процесів обробки інформації у людському мозку. Загальновідомо, що найбільший обсяг інформації обробляється через візуальний канал [1]. Це означає, що мозок виробив досить великий набір інструментарію, що дозволяє здійснювати уявні операції з візуальної інформацією. Це твердження підтверджується наявністю складних нейронних структур, утворених у нервовій системі людини для сприйняття та подальшої обробки візуальною інформацією [2].

Аналіз учбових матеріалів, призначений для використання учнями старших класів виявив недостатній рівень використання прийомів візуалізації при викладанні програмування. Так наприклад в затвердженому до використання на профільному рівні підручнику [3] виразні графічні елементи було використано лише при викладанні різних методів сортування даних. Навіть в розділі «Алгоритми на графах» ці суттєво візуальні алгоритми здебільшого представлені у текстовому вигляді. Але слід відзначити, що ми не пропонуємо модифікувати подібні підручники. Це призвело би до надмірного зростання їх об’єму. Більш доречним є розробка підтримуючих основний підручник методичних матеріалів, що роблять акцент на візуалізації.

Доречним прикладом такого ресурсу є [4], де використовується можливість використання інтеракції користувача (учня) з ресурсом.

Метою роботи є демонстрація типових випадків використання візуалізації в процесі вирішення задач програмування як засобу активізації пошукової та пізнавальної діяльності учнів.

Актуальність роботи у цьому напрямі обумовлена зростаючою потребою у фахівцях інформаційних технологій. Крім збільшення кількісної потреби, підвищується рівень складності розв'язуваних ними завдань. У цій ситуації пошук шляхів досягнення такого рівня досконалості видається вкрай важливим. Візуалізація як універсальний прийом представлення завдання та шляхів його вирішення є одним із ефективних механізмів. Тому у навчальному процесі важливе відпрацювання прийомів використання візуалізації. Далі розглянемо кілька завдань та проаналізуємо, наскільки візуальне уявлення сприяє їх вирішенню.

Задача 1: Визначити площу прямокутника шириною m та висотою n.

Це завдання є доволі простим прикладом «геометричного» завдання, де формулювання завдання провокує застосувати рішення за допомогою відомих у геометрії досить простих формул. Однак у нашому випадку цікавить вирішення цих завдань, що не використовує відповідних формул. В нашому випадку застосування формули S=m*n є найбільш природним. *Але ми на уроці програмування, і тому використовуємо інший підхід.* Наш прямокутник складається із квадратиків 1x1. Площа прямокутника – це кількість квадратиків.

Як підрахує площу програміст, який зовсім не знає геометричної формули (хоча у це важко повірити)? Програмістське розв’язання може виглядати так:

```
static void Main(string[] args)
{ int n=Convert.ToInt32(Console.ReadLine());
  int m = Convert.ToInt32(Console.ReadLine());
  int[,] rectangle = new int[n, m];
  int s=0;
  for (int row = 0; row < n; row++)
    for (int col = 0; col < m; col++)
      s++;
  Console.WriteLine(s);
}
```

Деякі зауваження до цієї простої програми.

Наш прямокутник представлений тут як двовимірний масив із ім'ям rectangle. Візуально його можна показати таким чином:

номери колонок (стовпчиків)
0 1 2 3
номери рядків
0
1
2
Двовимірний масив (матриця) rectangle

Рис. 1. Візуальне представлення двовимірного масиву

Використовуємо досить природні для звичайної людини терміни "рядки" (англійською rows) і "стовпці" (або "колонки", тому що англійською це columns). Не забуваймо, що в програмуванні нумерація елементів масиву, а також рядків та стовпців починається з 0.

На рис 1. зафарбована сірим кольором комірка знаходиться у стовпці 2 і рядку 1. У програмуванні ми дістанемося її за допомогою позначення rectangle[1,2]. Тут дуже важливо не переплутати – спочатку номер рядка, потім номер стовпця. Прийом для запам'ятовування – тут у програмуванні не так(!) як у геометрії. Якщо ми подивимося на рисунок як на ділянку площині з осями X і Y, то сіра «крапка» має координати X=2 Y=1 (вісь Y дивиться вниз!). Тобто геометрично це крапка з координатами (2;1). Але в програмуванні це rectangle[1,2]!

Наша програма – простий приклад програми знаходження суми.

Ми не використовуємо вміст комірок масиву, а лише підраховуємо їх кількість.

Для цього ми використовуємо «вкладений цикл», який зазвичай використовується для роботи з двовимірними масивами. Це конструкція, у якої головний цикл містить у собі внутрішній цикл. При кожному значенні індексу головного циклу внутрішній цикл просуває свій індекс від початку до кінця. Вся подорож по масиву візуально виглядає так:

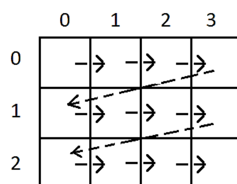

Рис. 2. Послідовність подорожі по матриці за допомогою вкладеного циклу

Для змінних у програмі слід використовувати зрозумілі імена. У нас це row, col та rectangle.

У чому полягає популярність завдань програмування, пов'язаних з одновимірними та багатовимірними масивами? Однією з причин є те, що суттєвою рисою багатьох предметних областей є присутність Евклідового простору з ортонормованим базисом. Таке уявлення простору є звичним і підтримується у межах шкільної геометрії.

У зв'язку з цим недивним є той факт, що практично у всіх процедурних мовах програмування основним контейнерним типом є масив. При цьому базова одновимірна версія масиву та його багатовимірні розширення синтаксично та семантично виглядають дуже схоже практично у всіх мовах програмування.

Більшість завдань програмування, в яких мається на увазі використання масивів, можуть бути природно представлені у візуальній формі. Основною ідеєю цієї статті є твердження про користь використання візуального подання при вирішенні таких завдань. Дидактичні основи цієї ідеї ґрунтуються на специфіці обробки інформації, властивій стандартному людському мисленню.

Самостійне завдання: Підрахувати площу прямокутника 4.5x5.5.

Задача 2: Реалізація гри в хрестики-нулики. Стан ігрового поля.

Якщо ми хочемо реалізувати програмну гру в хрестики-нулики, нам доведеться перевіряти на кожному кроці стан гри. Існують 4 варіанти:

- перемогли хрестики;
- перемогли нулики;
- нічия;
- можна продовжувати.

Ці варіанти можна закодувати якимось числами, але від цього програма стане менш зрозумілою. Тому для зрозумілості програми будемо використовувати для стану гри перерахований тип:

```
enum XOState { Xwin, Owin, Draw, Continue}
```

Саме ігрове поле природно змоделювати квадратним двовимірним масивом 3x3. Також для зрозумілості подання комірок ігрового поля використовуватимемо перерахований тип:

```
enum XOCell { X,O,Empty}
```

Тепер ігрове поле у програмі виглядатиме так:

```
XOCell[,] field;
```

Для перевірки роботи програми задамо якийсь стан ігрового поля, наприклад:

```
field = new XOCell[3, 3]
    { {XOCell.X, XOCell.O, XOCell.X},
      {XOCell.Empty,XOCell.O,XOCell.Empty},
      {XOCell.Empty,XOCell.Empty,XOCell.Empty}
    };
```

Для перевірки стану поля напишемо метод Result, який проаналізує який стан склався на полі.

Існує 8 варіантів перемоги одного з гравців, що показано на зображенні:

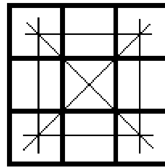

Рис. 3. Варіанти перемоги гравця в хрестики-нулики

Все просто – дві діагоналі, 3 рядки та 3 стовпці. Детальніше розглянемо головну діагональ:

```
if ((field[0,0]!=XOCell.Empty) &&
  (field[0,0]==field[1,1])&&(field[1,1]==field[2,2]))
   if (field[0,0]==XOCell.X) return XOState.Xwin;
   else return XOState.Owin;
```

Спочатку перевіряємо, що перший елемент діагоналі не порожній. Далі перевіряємо, що всі три елементи діагоналі рівні один одному.

Зауважте, що у мові C# не можна написати field[0, 0] == field[1, 1]) == field[2, 2]

Справа в тому, що операція == бінарна і повертає логічне значення. Перша операція == поверне true чи false. Далі цей true або false використовується у наступній операції ==. Але другий операнд має тип XOCell – несумісні типи, помилка!

Внутрішній умовний оператор уточнює, хто ж переміг, хрестики чи нулики.

Аналогічно перевіряємо решту з 7 можливих варіантів. В результаті виходить такий метод:

```
public static XOState Result(XOCell[,] field)
{
 //побічна діагональ
 if ((field[0,2] != XOCell.Empty) &&
   (field[0,2] == field[1,1]) && (field[1,1] == field[2,0]))
  if (field[0,2] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
 //головна діагональ
 if ((field[0,0] != XOCell.Empty) &&
   (field[0,0] == field[1,1]) && (field[1,1] == field[2,2]))
  if (field[0,0] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
 //перший рядок
 if ((field[0,0] != XOCell.Empty) &&
   (field[0,0] == field[0,1]) && (field[0,1] == field[0,2]))
  if (field[0,0] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
 //другий рядок
 if ((field[1,0] != XOCell.Empty) &&
   (field[1,0] == field[1,1]) && (field[1,1] == field[1,2]))
  if (field[1,0] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
 //третій рядок
 if ((field[2,0] != XOCell.Empty) &&
   (field[2,0] == field[2,1]) && (field[2,1] == field[2,2]))
  if (field[2, 0] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
 //перший стовпчик
 if ((field[0,0] != XOCell.Empty) &&
   (field[0,0] == field[1,0]) && (field[1,0] == field[2,0]))
  if (field[0,0] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
 //другий стовпчик
 if ((field[0,1] != XOCell.Empty) &&
   (field[0,1] == field[1,1]) && (field[1,1] == field[2,1]))
```

```
  if (field[0,1] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
//третій стовпчик
if ((field[0,2] != XOCell.Empty) &&
   (field[0,2] == field[1,2]) && (field[1,2] == field[2,2]))
  if (field[0, 2] == XOCell.X) return XOState.Xwin;
  else return XOState.Owin;
bool emptyCells = false;
for (int r = 0; r < 3; r++)
  for (int c = 0; c < 3; c++)
    if (field[r, c] == XOCell.Empty)
    {
      emptyCells = true;
      break;
    }
if (emptyCells) return XOState.Continue;
else return XOState.Draw;
}
```

Насамкінець метод, який визначає що ніхто не переміг, з'ясовує, чи можна продовжувати гру, тобто чи є на полі комірки XOCell.Empty.

Все гаразд. Така програма працює. Напишемо такий тест:

```
static void Main(string[] args)
{
  Console.OutputEncoding = Encoding.UTF8;
  XOCell[,] field;
  field = new XOCell[3, 3]
        { {XOCell.X, XOCell.O, XOCell.X},
          {XOCell.Empty,XOCell.O,XOCell.Empty},
          {XOCell.Empty,XOCell.Empty,XOCell.Empty}
        };
  switch(Result(field))
  {
    case XOState.Xwin:
      Console.WriteLine("перемога X"); break;
    case XOState.Owin:
      Console.WriteLine("перемога O"); break;
    case XOState.Draw:
      Console.WriteLine("нічия"); break;
    case XOState.Continue:
      Console.WriteLine("продовжуємо"); break;
  }
}
```

Тест видає правильний результат - "продовжуємо".

Але у стандартні хрестики-нулики грати нецікаво. Цікавіше, наприклад, грати на нескінченному полі (принаймні, розміром у листочок в клітинку) до 4-х поспіль хрестиків та нуликів.

І тут наш підхід призводить до дуже неефективного рішення. По-перше, тепер кількість переможних ситуацій вже не 8, а набагато більша. Перерахувати їх у програмному коді тепер буде практично неможливо. Крім того, Ваша інтуїція в цьому місці має підказати, що шукати перемогу чи поразку потрібно не на всьому великому полі, а поблизу останнього ходу. Усього ситуацій, які потрібно перевірити – 16. Вісім із них показано на рис. 4. Здогадайтеся, які 8 ситуацій на малюнку не показані.

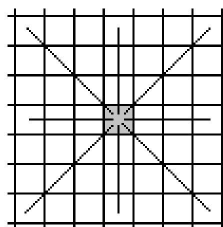

Рис. 4. Вісім варіантів перемоги з 16-ти

Ситуація ускладнюється тим, що така логіка перевірки трохи змінюється, якщо останній хід зроблено поблизу краю поля. Тоді потрібно перевіряти меншу кількість варіантів.

Наприклад, якщо хід зроблено у лівому верхньому кутку, то цікаві лише 3 варіанти (рис. 5).

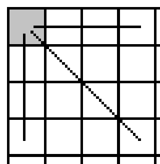

Рис. 5. Варіанти перемоги в куті ігрового поля

Але це для програміста дуже незручно – потрібно писати різний код для різних ситуацій, використовуючи складну сукупність умовних операторів. І тут вдалим будемо наступний досить популярний прийом – доповнення поля фіктивними значеннями.

У нашому випадку ми оточуємо все поле по периметру поясом з трьох рядків клітин, що містять значення XOCell.Empty. Тепер остання ситуація виглядатиме так, як на рис. 6.

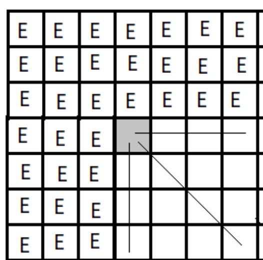

Рис. 6. Розширення ігрового поля фіктивним клітинками

Тепер перевіряючи околицю навколо точки останнього ходу (сіра клітинка), ми не боїмося вийти за межі ігрового поля. З рядків порожніх клітинок (заповнені буквою E) достатньо для цього, а значення XOCell.Empty у цих осередках не спотворює результат перевірки.

Для цього прийому напишемо допоміжний метод:

```
public static XOCell[,] ExtendField(XOCell[,] field)
{
  XOCell[,] extField= new XOCell[field.GetLength(0)+6, field.GetLength(1)+6];
  for (int row = 0; row < extField.GetLength(0); row++)
    for (int col = 0; col < extField.GetLength(1); col++)
      extField[row, col] = XOCell.E;
  for (int row = 0; row < field.GetLength(0); row++)
    for (int col = 0; col < field.GetLength(1); col++)
      extField[row+3, col+3] = field[row,col];
  return extField;
}
```

Тут ми резервуємо масив extField з урахуванням додаткових поясів (3 зверху, 3 знизу, 3 ліворуч і 3 праворуч). Далі заповнюємо весь масив XOCell.Empty (перший вкладений цикл). І нарешті, копіюємо масив field в розширений масив extField.

Тепер розглянемо основний метод перевірки стану гри. Крім ігрового поля він отримує інформацію про координати останнього ходу:

```
public static XOState Result(XOCell[,] field, int row, int col)
{
  //доповнюємо поле
  XOCell[,] extField = ExtendField(field);
  row = row + 2; col = col + 2;

  //розбираємо 16 ситуацій
  XOCell state;
  //головна діагональ 1
  state = FourEqual(extField[row-3, col-3],
           extField[row-2, col-2],
           extField[row-1, col-1],
```

```
                  extField[row, col]);
        if (state == XOCell.X) return XOState.Xwin;
        if (state == XOCell.O) return XOState.Owin;
        //головна діагональ 2
        state = FourEqual(extField[row-2, col-2],
                  extField[row-1, col-1],
                  extField[row, col],
                  extField[row+1, col+1]);
        if (state == XOCell.X) return XOState.Xwin;
        if (state == XOCell.O) return XOState.Owin;
        //головна діагональ 3
        state = FourEqual(extField[row-1, col-1],
                  extField[row, col],
                  extField[row+1, col+1],
                  extField[row+2, col+2]);
        if (state == XOCell.X) return XOState.Xwin;
        if (state == XOCell.O) return XOState.Owin;
        //головна діагональ 4
        state = FourEqual(extField[row, col],
                  extField[row+1, col+1],
                  extField[row+2, col+2],
                  extField[row+3, col+3]);
        if (state == XOCell.X) return XOState.Xwin;
        if (state == XOCell.O) return XOState.Owin;
        //аналогічно
        //побічна діагональ 1
        //побічна діагональ 2
        //побічна діагональ 3
        //побічна діагональ 4
        //рядок 1
        //рядок 2
        //рядок 3
        //рядок 4
        //стовпчик 1
        //стовпчик 2
        //стовпчик 3
        //стовпчик 4
        //якщо ми тут - ніхто не переміг
        bool emptyCells = false;
        for (int r = 0; r < 3; r++)
          for (int c = 0; c < 3; c++)
            if (field[r, c] == XOCell.E)
            {
              emptyCells = true;
              break;
            }
        if (emptyCells) return XOState.Continue;
        else return XOState.Draw;
      }
```

На початку розширюємо ігрове поле та перевірка стану далі відбувається вже у розширеному масиві.

Для зручності перевірки рівності 4-х осередків напишемо допоміжний метод FourEqual.

```
public static XOCell FourEqual(XOCell c1, XOCell c2, XOCell c3, XOCell c4)
{
  if ((c1==c2) && (c1 == c3) && (c1 == c4)) return c1;
  else return XOCell.E;
}
```

З його допомогою здійснюється 16 перевірок. У поданому програмному коді детально представлені перевірки, пов'язані лише з головною діагоналлю. Існує 4 діагоналі, що стосуються клітинки, яка нас цікавить:

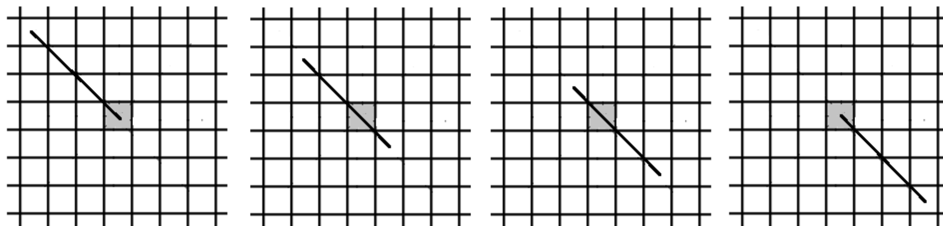

Рис. 7. Варіанти перемоги на головній діагоналі.

Самостійно: для повної перевірки слід реалізувати ще 14 ситуацій (див. коментарі у тексті програми).

Задача 3: Заповнення квадратної матриці «по спіралі».

Без зайвих слів завдання можна проілюструвати наступним рисунком:

1	2	3	4	5	6
20	21	22	23	24	7
19	32	33	34	25	8
18	31	36	35	26	9
17	30	29	28	27	10
16	15	14	13	12	11

Рис. 8. Заповнення матриці по спіралі

Таку задачу треба вирішити для будь-якого розміру квадрата N.

В чому особливість цієї задачі? Справа в тому, що більшість задач, пов’язаних з двовимірними масивами (матрицями) вирішується з використанням вкладених циклів, на кшталт того, що був використаний при розв’язанні задачі 1. Але послідовність проходження елементів масиву, яка показана на рис. 2, не дозволить вирішити завдання.

Поглянемо на задачу з точки зору поведінки мандрівника, який рухається по масиву, та залишає в клітинках числа. З цієї точки зору він або рухається в обраному напрямку, або повертає, якщо подальший рух вперед неможливий. Ця логіка реалізована в наступному коді:

```
static void Main(string[] args)
{ int N = 5;
  int[,] m= new int[N,N];
  int r = 0, c = 0, direction = 0, step = 1;
  m[r, c] = step;
  for (step = 2; step <= N*N; step++)
  { NextStep(ref r, ref c, m, ref direction);
    m[r, c] = step;
  }
  for (int i = 0; i < N; i++)
  { for (int j = 0; j < N; j++) Console.Write("{0:f2} ", m[i, j]);
    Console.WriteLine();
  }
}
```

Логіка руху «сховалась» в методі NextStep, тому метод Main доволі компактний.

Змінні r та c позначають координати наступної клітинки, dir – напрямок руху (0 – вправо, 1 – вниз, 2 – вліво, 3 – вверх), step – номер руху.

Перший крок робиться окремо. Останні – за допомогою циклу, який «знає», що таких рухів всього N*N.

Тепер головне – метод NextStep. Це цікавий різновид методу C#, який використовує свої параметри, змінює їх значення та повертає їх в Main. Для такого способу використання вони мають бути позначені ключовим словом ref.

```
static void NextStep(ref int r, ref int c, int[,] m, ref int dir)
{ int[] dr = { 0, 1, 0, -1 };
  int[] dc = { 1, 0, -1, 0 };
  int pr = r + dr[dir];
  int pc = c + dc[dir];
  if (pr<m.GetLength(0) && pc<m.GetLength(1) &&  pr>=0 && pc>=0 &&
      m[pr,pc]==0)  //можна продовжувати рух у напрямку
    r = pr; c = pc;
  else
  { dir = (dir + 1) % 4;
    r = r + dr[dir];
    c = c + dc[dir];
  }
}
```

Тут є три цікавих аспекти.

1) Послідовність зміни напрямків у числовому вигляді: 0,1,2,3,0,1,2,3, і т.д. Така послідовність виникає за рахунок вирахування остачі від ділення dir=(dir+1)%4;

2) Рух в напрямку dir доволі компактно реалізовано як r=r+dr[dir]; c=c+dc[dir]; Це виглядає компактно завдяки масиву dr, що задає наскільки змінюються координати клітинку в залежності від напрямку.

3) Перевірка того, чи потрібно повертати, це відбувається в умовному операторі.

Самостійно: поясніть, навіщо в NextStep знадобились допоміжні змінні pr та pc.

Таким чином у роботі продемонстровано низку прикладів розв'язання задач програмування, для вирішення яких корисно використовувати елементи візуалізації на етапах формулювання та розв'язання задачі. Можна відзначити два важливі аспекти, у яких візуалізація виявляється корисною. По-перше, візуалізація допомагає спілкуватися у процесі розв'язання завдання. По-друге, візуалізація активізує ментальні можливості програміста. У навчальній літературі такий прийом використовують досить мало. Проте, не можна стверджувати, що прийом з урахуванням візуалізації може бути стандартизований у межах загальноприйнятих підручників інформатики та програмування. Більш адекватним є гнучкий підхід, коли викладач використовує візуалізацію з огляду на контекст навчального процесу, власні індивідуальні особливості та особливості аудиторії учнів.

Список використаних джерел

1. *Vision Is Our Dominant Sense.* URL: https://www.brainline.org/article/vision-our-dominant-sense.
2. *Introduction to Psychology & Neuroscience.* URL: https://digitaleditions.library.dal.ca/intropsychneuro/chapter/vision/.
3. Руденко В.Д., Речич Н.В., Потієнко В.О. *Інформатика (профільний рівень): підручник для 11 класу.* Харків: Ранок, 2019. 256 с.
4. *Структури даних і алгоритми: Візуалізація деяких алгоритмів*. URL: https://sites.google.com/site/chnudatstruct/home/vizualizacia-deakih-algoritmiv

References

1. *Vision Is Our Dominant Sense.* URL: https://www.brainline.org/article/vision-our-dominant-sense.
2. *Introduction to Psychology & Neuroscience.* URL: https://digitaleditions.library.dal.ca/intropsychneuro/chapter/vision/.
3. Rudenko V.D., Rechych N.V., Potiienko V.O. *Informatyka (profilnyi riven): pidruchnyk dlia 11 klasu*. Kharkiv: Ranok, 2019. 256 s.
4. *Struktury danykh i alhorytmy: Vizualizatsiia deiakykh alhorytmiv.* URL: https://sites.google.com/site/chnudatstruct/home/vizualizacia-deakih-algoritmiv.