**Министерство образования и науки Российской Федерации**

**Пермское федеральное государственное бюджетное образовательное учреждение высшего образования «Пермский Национальный Исследовательский Политехнический Университет»**

**Электротехнический факультет**

**Кафедра «Информационные технологии и автоматизированные системы»**

**ОТЧЁТ**

По лабораторной работе №13 на тему

«Задача о восьми ферзях»

Вариант №11

Выполнил студент группы ИВТ-20-2б

Сабуров Павел Алексеевич

Проверил доцент кафедры ИТАС

Полякова О.А.

Пермь 2021

**Цель работы** –научить решать задачу о восьми ферзях, используя метод поиска с возвратом.

**Постановка задачи**

Задача – реализовать на языке C++ программу, решающую задачу о возьми ферзях с использованием метода поиска с возвратом, при этом необходимо:

* Вывести на экран каждый шаг решения – текущую расстановку ферзей на шахматном поле;
* Вывести на экран первое решение задачи;

**Анализ задачи**

Для решения задачи были использованы следующие средства:

1. Язык программирования C++ (Microsoft Visual C++)
2. Текстовый редактор Microsoft Visual Studio Code

Метод поиска с возвратом – это реализация метода прямого перебора элементов в массиве, в котором необходимо найти последовательность, удовлетворяющую определённым условиям. С начала подбирается первый элемент, затем второй, затем третий и так далее; если на определённой итерации не получается установить новый элемент, осуществляется возврат на предыдущий элемент, этот элемент нужно изменить так, чтобы появилась возможность установить следующий элемент.

Одной из задач, решаемой через поиск с возвратом является задача о восьми ферзях. Последовательность, которую необходимо найти – расстановка ферзей в шахматном поле таким образом, чтобы не один ферзь не стоял под боем другого ферзя.

Перебор в данной задаче устроен следующим образом:

1. Идёт перебор по строкам;
2. В каждой строке идёт перебор по столбцам в текущей строке;

Примечание: задачи с перебором последовательности – поиск с возвратом удобно решать с использованием рекурсии, именно с помощью неё будет осуществляться возврат в случае неудачной итерации.

**Реализация программы:**

Для операций ввода-вывода необходимо подключить заголовочный файл iostream, который предоставляет объект потока вывода std::cout. Заголовочные файлы подключаются при помощи директивы #include. Также, подключается пространство имён std;

Для удобства, при помощи директивы #define были определено два параметра:

1. Название задачи (для вывода в консоль);
2. Размер доски – 8 клеток;

#include <iostream>

#define APP\_TITLE "Eight queens puzzle"

#define FIELD\_SIZE 8

using namespace std;

В качестве шахматного поля будет выступать двумерный массив, содержащий целые числа. Для краткого объявления и инициализации в главной функции была реализована функция NewField, инициализирующая новый двумерный массив, заполняя каждый его элемент числом 0.

int\*\* field = new int\* [FIELD\_SIZE];

В функции создаётся новый динамический массив массивов (по смыслу тот же двумерный массив), который через цикл for заполняется новыми массивами, которые во вложенном цикле for заполняются числом 0.

for (int i = 0; i < FIELD\_SIZE; i++)

{

field[i] = new int [FIELD\_SIZE];

for (int j = 0; j < FIELD\_SIZE; j++)

field[i][j] = 0;

}

В конце выполнения функция возвращает заполненный массив.

return field;

Поскольку, после каждого шага придётся выводить шахматное поле на экран целиком, была реализована функция PrintField, которая выводит состояние шахматного поля в соответствии со значением числа клетки:

* 0 – поле свободное поле: фигура не установлена и не находится под боем ферзей;
* -1 – установлен ферзь;
* 1 и больше: фигура находится под боем ферзя, или нескольких ферзей;
* 0 – выводится .
* 1 и больше – выводится \*
* -1 – выводится Q

Поскольку, поиск с возвратом является перебором и подразумевает решение задачи путём большого количества шагов, было принято решение, в котором вместо вызова cout для каждого элемента – 8 \* 8 = 64 вызова cout на одно поле, создаётся строка, в которой накапливаются элемента, и в конце выводится эта «большая» строка через 1 cout.

Накапливание строки происходит через вложенные циклы for. Внешний цикл отвечает за строки, внутренний – за столбцы шахматного поля. Чтобы отделить одно поле от другого, в конце строки добавляется символ перехода на новую строку.

void PrintField(int\*\* chessField)

{

string field = "";

for (int i = 0; i < FIELD\_SIZE; i++)

{

for (int j = 0; j < FIELD\_SIZE; j++)

field += (chessField[i][j] == -1)? "Q " :

(chessField[i][j] > 0)? "\* " : ". ";

field += '\n';

}

field += '\n';

cout << field;

}

Как было описано выше, число 0 в данном массиве означает, что данная ячейка является свободной для размещения и не находится под боем других ферзей, если значение больше нуля – клетка под боем, -1 – стоит ферзь.

Соответственно, чтобы обозначить, что клетка стоит под боем ферзя, необходимо увеличить значение этой клетки, если ферзя убрали, то уменьшить, а если туда поставили ферзя, то присвоить -1, если убрали, то присвоить 0 (потому что ферзи будут ставиться только в те клетки, которые не стоят под боем других ферзей).

Получается, что функции, отвечающие за установку и удаление ферзя, будут работать абсолютно одинаково: будут отличаться только устанавливаемые значения. Поэтому была реализована общая функция, отвечающая за установку состояния шахматного поля SetState.

Для удобства и написания более читаемого кода, было добавлено перечисление Queen, в котором через Set или Delete будет определяться, что необходимо выполнить: установить или удалить ферзя. Также, в качестве аргумента принимается строка и столбец, в котором, будет происходить: установка или удаление ферзя.

void SetState(int\*\* chessField, Queen queenState, int row, int column)

В функции необходимо объявить две главные переменные, которые будут присвоены в соответствии с тем, что будет происходить с ферзём.

* Сдвиг: больше нуля – прибавляем, сигнализируем, что клетка под боем новоустановленного ферзя, меньше нуля, ферзя удалили, значит все клетки что, были под его боем возвращаются в предыдущее состояние.
* Значение клетки на позиции – -1: ферзь установлен, 0: ферзь удалён.

И через оператор switch, в соответствии с задачей присваиваются необходимые значения переменным.

int

offset = 0,

cellValue = 0;

switch (queenState)

{

case Queen::Set:

offset = 1;

cellValue = -1;

break;

case Queen::Delete:

offset = -1;

cellValue = 0;

break;

}

Затем, необходимо самим клеткам присвоить необходимые значения. Это будет происходить через цикл for. Ферзь бьёт по горизонтали, по вертикали и по диагонали.

for (int i = 0; i < FIELD\_SIZE; i++)

{

//Заполнение клеток

}

Заполнение по горизонтали:

chessField[row][i] += offset;

Заполнение по вертикали:

chessField[i][column] += offset;

Заполнение по диагонали:

int foo = column - row + i;

if (foo >= 0 && foo < FIELD\_SIZE)

chessField[i][foo] += offset;

foo = column + row - i;

if (foo >= 0 && foo < FIELD\_SIZE)

chessField[i][foo] += offset;

В конце выполнения функции: устанавливается значение клетки и поскольку было изменено состояние шахматного поля, то есть произведён шаг, осуществляется вывод нового полученного поля на экран.

chessField[row][column] = cellValue;

PrintField(chessField);

Как было описано выше, алгоритм поиска с возвратом удобно решать при помощи рекурсии. Каждая функция отвечает за установку-удаление ферзей на каждой строке, через проверку значений клеток каждого столбца данной строки. Если функция возвращает true – то есть ферзя поставить можно, то вызывается вложенная функция, если та возвращает true, то в свою очередь вызывает вложенную функцию. Так будет происходить до тех пор, пока не будет достигнута последняя строка в шахматной доске.

bool CanSetQueen(int\*\* chessField, int row)

{

bool result = false;

for (int column = 0; column < FIELD\_SIZE; column++)

//перебор всех столбцов строки

//возвращение значения: можно ли поставить ферзя или нет?

return result;

}

При переборе, функции достаточно лишь проверить свободна ли текущая клетка или нет (0: клетка свободна). Если клетка свободна, то можно поставить ферзя. Если это не последняя строка, то необходимо поставить следующего ферзя. Если ферзя поставить не удаётся, то ферзь удаляется, а перебор продолжается.

if (chessField[row][column] == 0)

{

SetState(chessField, Queen::Set, row, column);

if (row == FIELD\_SIZE - 1)

result = true;

else

{

result = CanSetQueen(chessField, row + 1);

if (!result)

SetState(chessField, Queen::Delete, row, column);

}

}

Также была реализована функция ShowSolution, которая принимает в качестве аргумента поле, в котором происходит поиск, и отвечающая за показ ответа или его отсутствия, которая будет вызываться в главной функции программы.

Если последовательность найти не получилось, то выведется соответствующее о провале поиска нужной комбинации на поле.

void ShowSolution(int\*\* field)

{

bool hasSolution = CanSetQueen(field, 0);

if (hasSolution)

{

cout << "Final layout:" << endl;

PrintField(field);

}

else

cout << "The task has no solutions.";

}

В главной функции происходит следующее:

* Выводится на экран заголовок программы;
* Инициализируется новый двумерный массив;
* Вызывается функция, отвечающая за решение задачи;

int main()

{

cout << APP\_TITLE << endl;

int\*\* chessField = NewField();

ShowSolution(chessField);

delete[] chessField;

return 0;

}

**Полный исходный код программы на языке программирования C++:**

#include <iostream>

#define APP\_TITLE "Eight queens puzzle"

#define FIELD\_SIZE 8

using namespace std;

enum Queen

{

Set, Delete

};

int\*\* NewField()

{

int\*\* field = new int\* [FIELD\_SIZE];

for (int i = 0; i < FIELD\_SIZE; i++)

{

field[i] = new int [FIELD\_SIZE];

for (int j = 0; j < FIELD\_SIZE; j++)

field[i][j] = 0;

}

return field;

}

void PrintField(int\*\* chessField)

{

string field = "";

for (int i = 0; i < FIELD\_SIZE; i++)

{

for (int j = 0; j < FIELD\_SIZE; j++)

field += (chessField[i][j] == -1)? "Q " :

(chessField[i][j] > 0)? "\* " : ". ";

field += '\n';

}

field += '\n';

cout << field;

}

void SetState(int\*\* chessField, Queen queenState, int row, int column)

{

int

offset = 0,

cellValue = 0;

switch (queenState)

{

case Queen::Set:

offset = 1;

cellValue = -1;

break;

case Queen::Delete:

offset = -1;

cellValue = 0;

break;

}

for (int i = 0; i < FIELD\_SIZE; i++)

{

chessField[i][column] += offset;

chessField[row][i] += offset;

int foo = column - row + i;

if (foo >= 0 && foo < FIELD\_SIZE)

chessField[i][foo] += offset;

foo = column + row - i;

if (foo >= 0 && foo < FIELD\_SIZE)

chessField[i][foo] += offset;

}

chessField[row][column] = cellValue;

PrintField(chessField);

}

bool CanSetQueen(int\*\* chessField, int row)

{

bool result = false;

for (int column = 0; column < FIELD\_SIZE; column++)

if (chessField[row][column] == 0)

{

SetState(chessField, Queen::Set, row, column);

if (row == FIELD\_SIZE - 1)

result = true;

else

{

result = CanSetQueen(chessField, row + 1);

if (!result)

SetState(chessField, Queen::Delete, row, column);

}

}

return result;

}

void ShowSolution(int\*\* field)

{

bool hasSolution = CanSetQueen(field, 0);

if (hasSolution)

{

cout << "Final layout:" << endl;

PrintField(field);

}

else

cout << "The task has no solutions.";

}

int main()

{

cout << APP\_TITLE << endl;

int\*\* chessField = NewField();

ShowSolution(chessField);

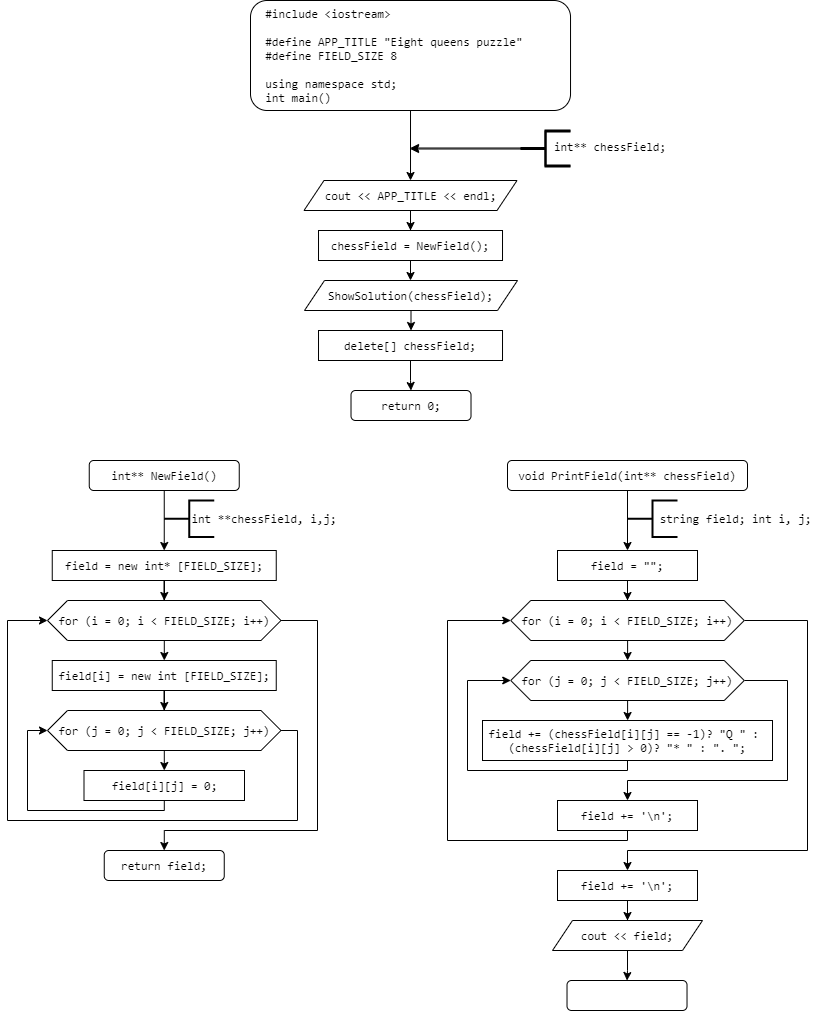
delete[] chessField;

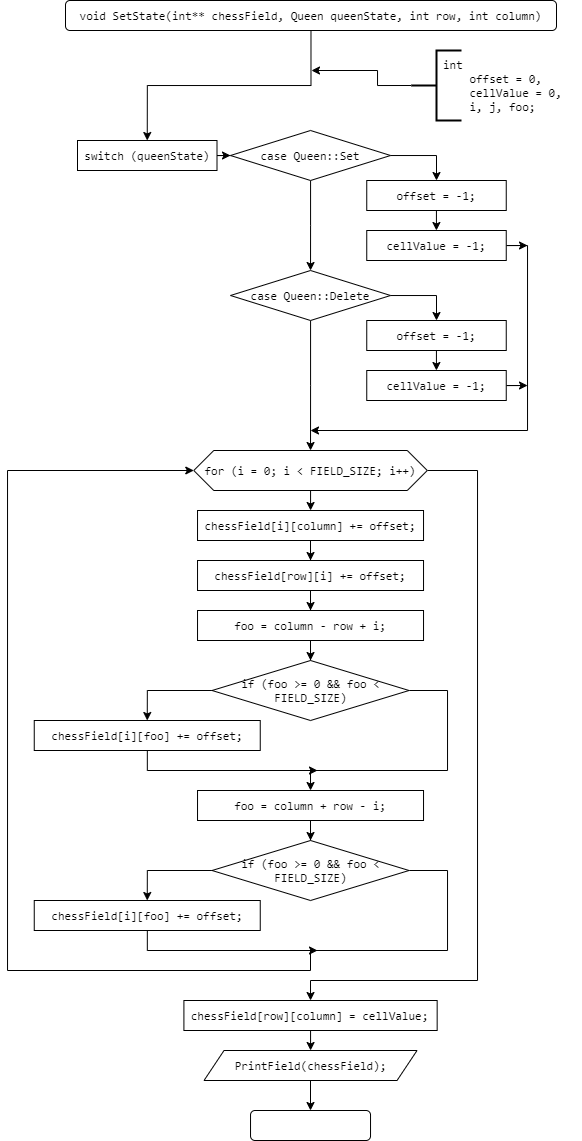
system("pause");

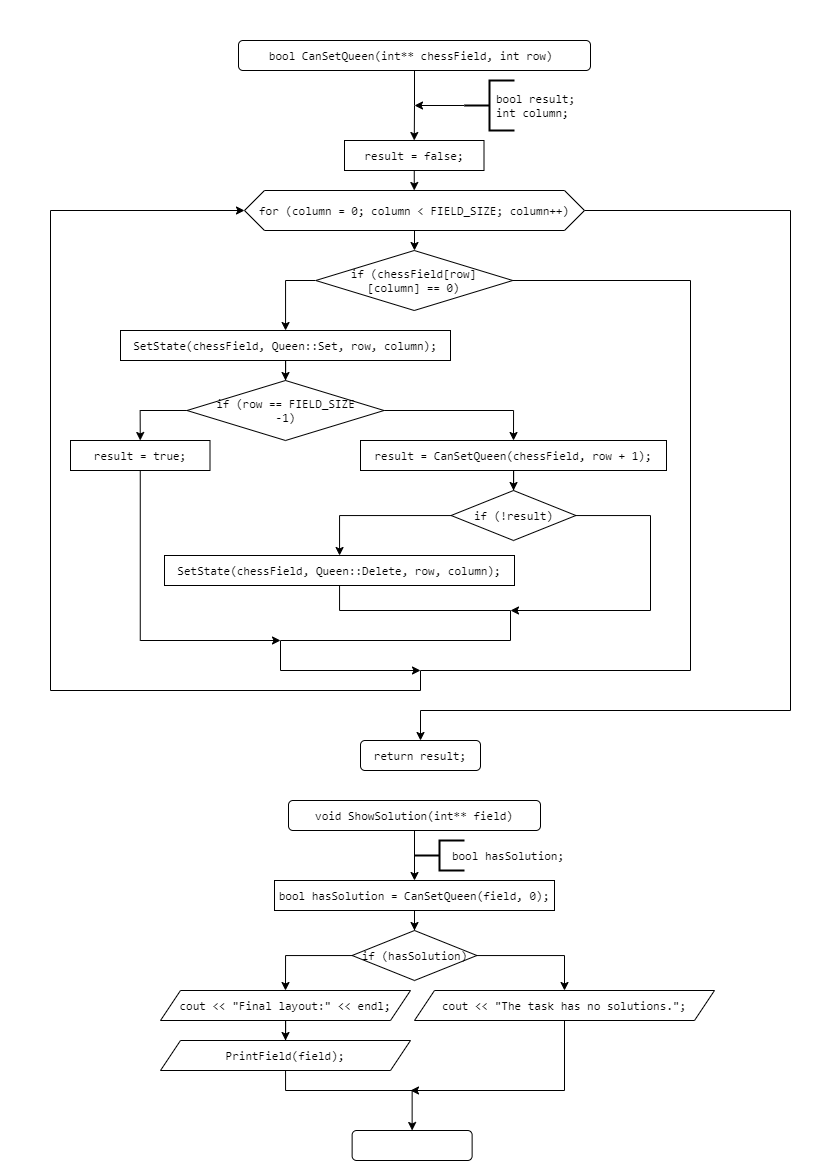
return 0;

}

**Блок-схема** алгоритма программы:







**Скриншоты выполненной программы:**

