Politechnika Śląska w Gliwicach

Wydział Informatyki, Elektroniki i Informatyki

Programowania Komputerów 4

Warcaby

autor	Dorian Barański
prowadzący	mgr inż. Krzysztof Pasterak
rok akademicki	2018/2019
Kierunek	informatyka
rodzaj studiów	SSI
semestr	4
termin laboratorium / ćwiczeń	wtorek, 14:00 – 15:30
grupa	6
sekcja	1
Data oddania sprawozdania	2019-06-24

1. Temat

Warcaby – gra w trybie tekstowym dla dwóch graczy z wykrywaniem poprawności ruchów.

2.1 Struktura danych

Program został stworzony obiektowo, obiekty stworzone w programie mają swoją klasę. Gra oparta jest na wektorach, czyli tablicach dynamicznych. Szachownica pełni rolę wektora wektorów i przechowuje wskaźniki na figury. Dodatkowo każdy gracz posiada wektor wskaźników na swoje bierki, w zależności od ich koloru.

W warcabach obowiązują zasady, związane z poruszaniem się figurami po szachownicy w ściśle określony sposób i zależny od tego czy ruch jest wykonywany pionkiem czy też królową.

2.2 Analiza problemu

Program wymaga od użytkowników stosowania się do narzuconych przez grę zasad i nie akceptuje nieprawidłowych ruchów, które są automatycznie pomijane, a program oczekuje wówczas na podanie ruchów zgodnych z zasadami.

Obowiązujące zasady:

- Pionek może poruszać się o jedno pole do przodu po przekątnej na wolne pola.
- Bicie pionkiem następuje przez przeskoczenie pionka lub damki przeciwnika na pole znajdujące się tuż za nim po przekątnej. To pole musi być wolne.
- Pionki mogą bić do przodu i do tyłu. Jest możliwe bicie wielokrotne podczas jednego ruchu.
- Damkę zdobywa się gdy pion dojdzie do ostatniego rzędu planszy.
- Po awansie kolejny ruch należy do przeciwnika
- Damki mogą poruszać się w jednym ruchu o dowolną liczbę pól do przodu lub do tyłu po przekątnej, zatrzymując się na wolnych polach.
- Bicie damką jest możliwe w dowolnej odległości po linii przekątnej i następuje przez przeskoczenie figury przeciwnika, za którym musi znajdować się co najmniej jedno wolne pole, można kontynuować bicie wzdłuż linii prostopadłej.

3.Specyfikacja zewnętrzna

Po uruchomieniu programu jest on gotowy do gry. Zostanie wyświetlona szachownica z ustawionymi pionkami obu użytkowników. Pierwszy ruch podczas nowej rozgrywki należy do białych figur. Program został podzielony na 6 plików nagłówkowych ".h" oraz 6 plików źródłowych ".cpp".

3.1 Ruchy

Program wykonuje ruchy figur po przez odczytanie wprowadzonych przez użytkowników komend. Składają się z dwóch znaków. Pierwszy to litery z zakresu A-H (przy domyślnej wielkości szachownicy 8x8), które odpowiadają kolumnom, a drugi to liczby 1-8 czyli numery wierszy. Ruchy muszą być oddzielone spacją. Pierwsza komenda odpowiada za wybór figury. Następna odnosi się już do przemieszczenia bierką. Wielkość liter nie ma znaczenia.
Przykładowy ruch:

A3 B4

- *A3* chwyta za pionek na tym polu i przesuwa po prawym skosie o jedno pole na *B4*.

Przykładowy ruch bicia pionkiem:
A3 C5

Wielokrotne bicie:
A3 C5 E3

4. Specyfikacja wewnętrzna

4.1 Class'y i struktura

W programie zostało zdefiniowanych pięć klas oraz jedną strukturę.

```
class Gameboard
{
public:
	Gameboard(int boardSize = 8, int rowOfPawns = 3);
	void play();
	int getSize();
	void movePiece(int source_file, int source_rank, int target_file, int
target_rank);
	void removePiece(int file, int rank);
	void promotePawn(int file, int rank);
	void copyRealToTempBoard();
	void copyTempToRealBoard();
	whatsStandingThere whatStandsThere(int file, int rank);
	~Gameboard();

private:
	void print();
	void waitForCorrectMove();
	bool makeMoveIfCorrect();
	void changePlayerToMove();
	bool checkForAnyPossibleMoves();
	bool tryToParse(std::string move_line);

	std::vector<Position> actualMove;
	const int boardSize;
	const int rowOfPawns;
	std::vector<std::vector<Square>> board;
	std::vector<std::vector<Square>> board_temp;
	std::vector<Piece*> whitePieces;
	std::vector<Piece*> blackPieces;
	Color whoseMove;
};
```

Klasa *Gameboard* reprezentuje szachownice o domyślnych parametrach. Służy temu zdefiniowany konstruktor. Inicjalizuje dwie szachownice, główną i pomocniczą(temp) oraz ustawia pionki. Wszelkie ruchy są wpierw testowane na *board_temp* , w celu sprawdzenia poprawności komend i możliwości ruchu. Jeżeli wszystko przebiegło pomyślnie wówczas główna szachownica jest nadpisywana przez szachownicę pomocniczą. Ta klasa posiada również metody odnoszące się do przesuwania, usuwania i promocji figur.

Posiada pola prywatne, które w większości to metody odnoszące się do poprawności ruchów (oczekiwanie na wprowadzenie, sprawdzenie czy jest możliwy ruch, przekazanie ruchu drugiemu graczowi) oraz wektory (tablice dynamiczne) czyli struktury danych naszego programu.

```
class Piece
{
public:
	Piece(Gameboard * motherboard_, int file, int rank, Color
pieceColor_);
	Color getPieceColor();
	virtual bool canMove() = 0;
	virtual bool makeSingleStep(Position source, Position target) = 0;
	virtual void printPiece() = 0;
	virtual ~Piece();

protected:
	Gameboard * motherboard;
	Color pieceColor;
	int rank;
	int file;
};
```

Klasa *Piece* to klasa bazowa. Dziedziczy po niej klasa *Pawn* oraz *Queen*. Konstruktor tworzy bierkę na szachownicy o odpowiednich współrzędnych i o odpowiednim kolorze. Posiada metody wirtualne z tego względu, że na szachownicy przechowujemy wskaźniki do bierek i to na ich rzecz wywołujemy metodę np. *makeSingleStep();* a że nie wiemy czy w danej sytuacji bierka = pionek czy bierka = królowa to ta metoda musi być wirtualna aby sama się dopasowała. Wówczas dla pionka ma wywołać:
Pawn->makeSingleStep();
A dla królowej:
Queen->makeSingleStep();
Metody wirtualne z pewnością nie wpływają pozytywnie na szybkość działania programu, lecz zdecydowanie upraszczają kod.

```
class Pawn : public Piece
{
public:
	Pawn(Gameboard * motherboard_, int file, int rank, Color
pieceColor_);
	bool makeSingleStep(Position source, Position target) override;
	void printPiece() override;
	bool canMove() override;
	virtual ~Pawn();
};
```

Klasa *Pawn* czyli pionek jest klasą pochodną, dziedzicząca po klasie *Piece*. Konstruktor tworzy pionek na szachownicy o zadanych współrzędnych i odpowiednim kolorze. Dziedziczy funkcje wirtualne, odpowiadające za przemieszczanie się, destrukcję i wydrukowanie pionków w danym kolorze.

```
class Queen : public Piece
{
public:
	Queen(Gameboard * motherboard_, int file, int rank, Color
pieceColor_);
	bool makeSingleStep(Position source, Position target) override;
	void printPiece() override;
	bool canMove() override;
	virtual ~Queen();
};
```

Klasa *Queen* podobnie jak *Pawn*, jest klasą pochodną, dziedziczącą po klasie Piece. Konstruktor tworzy na szachownicy figurę królowa o zadanych współrzędnych i o odpowiednim kolorze. Dziedziczy metody wirtualne odpowiedzialne za poprawność ruchu, wydrukowanie i destrukcje figury.

```
class Square
{
public:
	Square(Color color_);
	Color getSquareColor();
	Piece * getPiece();
	void removePiece();
	void setPiece(Piece * newPiece);
	~Square();
private:
	const Color squareColor;
	Piece * piecePointer = nullptr;
};
```

Klasa *Square* reprezentuje pojedyncze pole na szachownicy. Ustawia figurę na określonym miejscu, ściąga ją lub zwraca. Bierki stoją tylko na czarnych polach.

Klasa posiada również pola prywatne, czyli zmienną przechowującą kolor danego pola oraz wskaźnik na bierkę, która na danym polu stoi.

```
struct Position
{
	int file;
	int rank;
	Position(int file_, int rank_) : file{ file_ }, rank{ rank_ }{};
};
```

Ta struktura reprezentuje pojedynczy punkt na szachownicy. Zmienna *file* odpowiada kolumnom, iterowanym od lewej do prawej. Natomiast *rank* odpowiada rzędom, od dołu do góry.

4.2 Szczegółowy opis implementacji funkcji

`void Gameboard::play()`

Funkcja rozpoczyna rozgrywkę. Dopóki aktualny gracz ma możliwy ruch drukowana jest szachownica, program oczekuje na wprowadzenie poprawnego ruchu, po wykonaniu przekazuje ruch drugiemu graczowi. Po zakończeniu drukuje jeszcze raz szachownice i komunikat, że nie ma możliwości ruchu dla danego gracza.

`whatsStandingThere Gameboard::whatStandsThere(int file, int rank)`

Funkcja sprawdza co stoi na polu o danych współrzędnych.
-Jeżeli podane współrzędne są niepoprawne wówczas zwraca wartość *OutOfRange*.
-Jeśli pole jest puste - zwraca wartość *emptySquare*.
-Jeżeli pole jest zajęte – zwraca kolor bierki, która na tym polu stoi.

`bool Gameboard::checkForAnyPossibleMoves()`

Sprawdza czy aktualny gracz ma jakikolwiek możliwy ruch do wykonania. Przechodzi po wszystkich bierkach gracza białego, jeżeli jakaś figura może się ruszyć wtedy przerywa pętlę i zwraca wartość *true*. Dla gracza czarnego działa to w takim sam sposób. Jeżeli pętle nie znalazły żadnej bierki, która może się ruszyć, zwraca *false*.

`bool Gameboard::tryToParse(string move_line)`

Funkcja otrzymuje za argument linię, którą wprowadził gracz. Tworzy z niej *stringstream*, z którego można łatwo wyciągać pojedyncze słowa. Dla każdego wyciągniętego słowa w pętli sprawdza czy pierwszy znak jest literką z przedziału A-H i czy drugi znak to liczba z przedziału 1-8. Czynność powtarza w celu przerobienia całej linii podanej przez użytkownika. W przypadku powodzenia, funkcja dodaje sczytaną pozycję do wektora pozycji, który reprezentuje pojedynczy ruch.

```
Gameboard::Gameboard(int boardSize_, int rowOfPawns_) : boardSize{
boardSize_ }, rowOfPawns{ rowOfPawns_ }
```

Konstruktor klasy *Gameboard*. Inicjalizuje szachownicę główną oraz pomocniczą. Ustawia pionki białe oraz czarne na odpowiednich miejscach oraz sprawdza poprawność rozmiarów szachownicy i ilości rzędów z pionkami.

```
void Gameboard::waitForCorrectMove()
```

Funkcja sprawdza kolejne ruchy wprowadzane przez gracza do momentu aż będą one poprawne (zgodne z zasadami gry).

W celu spełnienia kryteriów wyznaczonych do zaliczenia projektu wykorzystałem w programie:

1. **Iterator**

   w funkcji:

```
void Gameboard::print(){
.
.

for (int i = boardSize - 1; i >= 0; i--)
{
      cout << setw(2) << i + 1 << " ";
      for (auto it = board[i].begin(); it != board[i].end();
++it) //iterator
      {
            if (auto piece = (*it).getPiece())
            {
                  piece->printPiece();
            }
            else
                  cout << " ";
            cout << " ";
      }
      cout << i + 1 << " \n";
}
```

   Ułatwił i usprawnił pracę na wektorach.

2. **Kontenery (vectory)**

```
std::vector<Position> actualMove;
std::vector<std::vector<Square>> board;
std::vector<std::vector<Square>> board_temp;
std::vector<Piece*> whitePieces; //wektor bierek bialego
std::vector<Piece*> blackPieces; //wektor bierek czarnego
```

   Dynamiczne tablice, czyli pojemniki na dane. Został na nich oparty cały program.

3. **Regex**

   w funkcji:

```
bool Gameboard::tryToParse(string move_line)
{

actualMove.clear();
regex line_regex{ "^[a-zA-Z0-9 ]*$" }; //regex
if (!regex_match(move_line, line_regex))
{
      cout << "string zawiera znaki specjalne" << endl;
      return false;
}
```

Sprawdza czy string wprowadzany przez gracza posiada znaki specjalne, jeżeli tak zwraca false i komentarz o błędzie.

4. **Wyjątek**

   w funkcji:

```
Gameboard::Gameboard(int boardSize_, int rowOfPawns_) :
boardSize{ boardSize_ }, rowOfPawns{ rowOfPawns_ }
{
      if (boardSize_ < 8 || boardSize_ > 20) throw
exception("Niewymiarowa szachownica."); //wyjatek
      if (rowOfPawns_ < 1 || rowOfPawns_ >= (boardSize_ / 2))
throw exception("Niepoprawna liczba pionow."); //wyjatek
```

Sprawdza czy wprowadzono prawidłową wielkość szachownicy oraz ilość rzędów z pionkami. Jeżeli wykryto błąd, wówczas *„wyrzuca”* wyjątek. Wywoływany w *main*:

```
try
{
      Gameboard first{ BOARD_SIZE, ROW_OF_PAWNS };
      first.play();
}
catch (exception ex)
{
      cout << ex.what() << endl;
}
```

5.Ogólna idea działania programu

Program drukuje szachownice o przyjętych wymiarach (domyślnie 8x8), czyli kolumny A-H oraz rzędy 1-8, następnie umieszcza na szachownicy rzędy pionków, ich ilość również można dowolnie zmieniać. Po wydrukowaniu program oczekuje na wprowadzenie komendy przez użytkownika grającego figurami białymi. Linia komend testowana jest wpierw na szachownicy pomocniczej, jest to niewidoczne dla użytkownika. Gdy ruch jest poprawny wówczas się wykonuje, w przeciwnym wypadku program pomija wprowadzoną komendę i oczekuje na wprowadzenie kolejnej, poprawnej. Po wykonaniu, ruch jest przekazywany graczowi drugiemu.

6. Testowanie

Program został przetestowany wielokrotnie w celu udoskonalenia programu i usunięcia ewentualnych błędów. Wprowadzenie błędnych komend czyli:

-zła konstrukcja ruchu
-chwytanie za figurę przeciwnika
-wykonanie ruchu na zajęte pole

Spowoduje pominięcie przez program wczytanych od gracza linii i wprowadzi grę w proces oczekiwania na poprawny ruch.

7. Wnioski

Program Warcaby wydawał się dość prosty w realizacji, lecz potrzeba sprawdzania wszelkich możliwych do wykonania ruchów znacznie utrudniła wykonanie projektu. Jest możliwość udoskonalenia gry po przez dodanie np. wyświetlania zakresu ruchu dla każdej figury na szachownicy. Program jest w pełni funkcjonalny. Posiada możliwość promocji pionka do królowej oraz wielokrotne bicie każdą figurą. W rozpoczynanych pojedynkach pierwszeństwo ruchu mają figury białe, zgodnie z zasadami gry. W wykonaniu projektu bardzo pomogła mi wiedza zaczerpnięta na laboratoriach i umożliwiła dostosowanie programu do wymagań przedmiotu.

Literatura:

Jerzy Grębosz. Symfonia C++ standard. Wydawnictwo, EDITION 2000, Kraków, 2000.