
# Object-oriented C++ as a tool for creating board games: a game of Blokus

Wincenty Szymura

9986004

School of Physics and Astronomy

The University of Manchester

May 2020

**Abstract**

Object-oriented C++ was used in order to create a game of Blokus. The implementation of the strategic game uses a variety of standard tools provided by the language and associated with the programming paradigm, as well as diverse advanced features. The game has full functionality of the original and a user-friendly interface. The latter, however, was achieved with the help of the windows.h library making the game Windows exclusive.

# 1 Introduction

Blokus is a French board game designed to be played by four players. Each player is given a set of 21 pieces of a different, single colour at the beginning of the game. The game is played in turns and consists of placing the pieces on a board. All the pieces are free polyominoes [1]; geometric figures made out of a number of squares. In the case of this game, the number varies between one and five. The board is also made out of squares, and each side has a length of 20. The pieces can only be placed on empty board squares and cannot touch other pieces belonging to the same player with any of their sides. They have to, however, touch a corner of at least one such piece with one of their corners. The game is played until none of the players can place any of their pieces on the board, or all the players have used all of their pieces. The goal of each player is to occupy the biggest area of the board by the end of the game [2].
The game is a two-dimensional strategic board game requiring both, users' input and relatively advanced visual output. There is no random element, such as a dice roll, involved. The game involves 84 pieces, four players and one board, in total. Mechanics of the game include manipulation of the pieces while they are still in player's possession and placing them on the board.

# 2 Code design and implementation

## 2.1 Code structure

The program contains a total of 19 classes, out of which 17 are classes for pieces. A graphical representation of the `piece` classes hierarchy is displayed in Figure 1. The five derived classes situated on the second level of the hierarchy are also abstract and each of them has at least one derived child class. There is a total of 11 such child classes and those are not abstract. The abstract derived classes differ between each other in the number of squares making up the piece, while the non-abstract derived classes one level below differ in dimensions of the smallest rectangle which can enclose the piece. Such distinction is necessary due to the structure of the `piece` objects. The rectangle is an essential component used to define each of them, as well as all their mechanics. The entirety of the `piece` class member data, as well as the member functions are shown in Figure 2. `is_there_square` member is an array holding information about which square positions within the rectangle of dimensions `length` and `width` are occupied by the squares making up the `piece`. Each `piece` can be rotated or flipped, while still in player's possession, using member functions `rotate` and `flip`, respectively. It can also be picked up in order to be placed on the board. Those functions are all pure virtual as the methods vary between specific `piece` classes.

The five non-virtual functions share their methods between the derived classes.

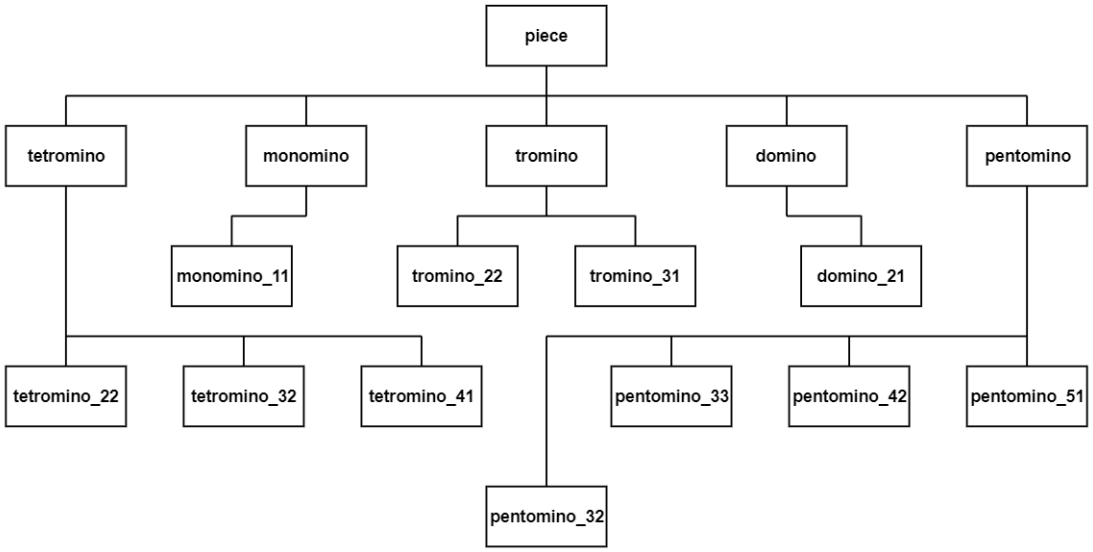


Figure 1: The abstract parent class called `piece` and all the derived classes presented in form of a tree graph. The hierarchy makes use of all three base concepts of object-oriented programming: encapsulation, inheritance and polymorphism.

```
class piece // abstract parent class
{
protected:
    size_t number_of_squares;
    size_t length; // the horizontal dimension of the piece
    size_t width; // the vertical dimension of the piece
    size_t handle; // defines which square is used to place the piece on board
    std::unique_ptr<bool[]> is_there_square;
public:
    piece();
    piece(const size_t squares, const size_t l, const size_t w,
        const size_t hand);
    piece(const piece&); // copy constructor
    piece(piece&&) noexcept; // move constructor
    virtual ~piece() {}
    piece& operator=(const piece&); //copy assignment
    piece& operator=(piece&&) noexcept; //move assignment
    // access an is_there_square element
    bool& operator()(const size_t m, const size_t n) const;
    size_t get_number_of_squares() const;
    size_t get_length() const;
    size_t get_width() const;
    size_t get_handle() const;
    virtual void rotate() = 0;
    virtual void flip() = 0;
    virtual void pick_up(std::vector<std::string>& directions) const = 0;
    void display(const size_t x_left, const size_t y_top,
        const size_t signature) const;
    //x_left and y_top define corner of rectangle the piece can be fitted into
};
```

Figure 2: The declaration of the abstract parent `piece` class. The class contains five members, four constructors, a virtual destructor, two overloaded operators and eight functions, including three pure virtual ones.

The two other classes, `players_set` class and `board` class, have a different relationship

with the family of `piece` classes. They are both composite classes, the former contains a vector of `piece`'s, and more specifically of base class pointers, while the latter contains a vector of `players_set`'s. Their declarations are shown in Figures 3 and 4, respectively. The main goal of instances of the `players_set` class is to monitor the number of `piece`'s left in the players' possession, and to facilitate and organise interactions between them and the `board`. Additionally, the class plays an important role in the graphical presentation of the game, holding two console coordinate vectors (`coordinates_horizontally` and `coordinates_vertically`), the `colour` associated with the player, and the `display`, `display_piece` and `erase_piece` member functions. The coordinate vectors are necessary due to limited space on the console which leaves some pieces in need of relocation upon being rotated.

```
class players_set // class containing all the pieces still in possession of a player
{
private:
    std::string colour;
    std::vector<std::shared_ptr<piece>> set_of_pieces;
    // vectors holding console coordinates used for displaying the set
    std::vector<std::pair<size_t, size_t>> coordinates_horizontally;
    std::vector<std::pair<size_t, size_t>> coordinates_vertically;
public:
    players_set() {}
    players_set(const std::string players_colour);
    ~players_set() {}
    std::string get_colour() const;
    size_t get_number_of_pieces() const;
    std::shared_ptr<piece> get_piece(const size_t piece_number) const;
    void remove_piece(const size_t piece_number); // after being placed on the board
    void display() const; // the entire set of pieces
    void erase_piece(const size_t piece_number) const; // from the console window
    void display_piece(const size_t piece_number) const; // on the console window
};
```

Figure 3: The declaration of the `players_set` class. The default constructor initializes an empty and colourless set. The parametrized one associates a `colour` with the player and appends 21 objects to each of the vectors.

The `board` class, which has the widest functionality, is responsible for the mechanics of the game. Of main importance is the `take_turn` function which explicitly or implicitly uses the majority of the other member functions. The `check_if_can_place` function imposes conditions for players' further participation in the game, eventually leading to its end, and is called separately in `main` before every turn. The `board` only has a default constructor, as there is only one way to initialize a Blokus board.

```
class board
{
private:
    size_t size; // one-dimensional length of the board
    size_t number_of_players;
    std::vector<players_set> players;
    std::unique_ptr<std::string[]> board_colours;
    static const std::vector<char> letters; // first 20 letter of the alphabet
    static const std::vector<std::string> possible_colours; // for players
public:
    board(); // there is only one way to set up the board
    size_t get_size() const;
    size_t get_number_of_players_left() const;
    players_set get_player(const size_t player_number) const;
    size_t get_players_score(const size_t player_number) const;
    std::string get_players_colour(const size_t player_number) const;
    size_t convert_input_to_position(const char board_letter,
        const size_t board_number) const;
    bool check_board_square(const size_t player_number,
        const size_t position) const; // and adjacent squares
    bool check_diagonally(const size_t player_number,
        const size_t position) const; // squares diagonally wrt position
    // is used from the second round onward
    bool place_piece(const size_t player_number, const size_t piece_number,
        const size_t position, const bool test, const bool players_test);
    bool place_first_piece(const size_t player_number,
        const size_t piece_number); // in the first round
    void print_piece_menu(const bool capital) const;
    void print_option_menu(const bool capital) const;
    void print_place_menu(const bool capital) const;
    void erase_text(const size_t x, const size_t y) const;
    void take_turn(const size_t player_number, const bool first);
    bool find_available_moves(const size_t player_number,
        const size_t piece_number, const bool players_test);
    bool check_if_can_place(const size_t player_number); // any piece
    void remove_player(const size_t player_number); // when they lose
    void display() const; // the board
};
```

Figure 4: The `board` class contains all the board parameters, a vector of players and two static constant vectors used for construction and in some of the functions. There are 18 member functions.

All the above-mentioned classes are divided between four pairs of header and source files. Both, the `board` class and the `players_set` class, have their own pairs. In case of the `piece` classes, it was decided that file handling complexity arising from the number of files required to store all the classes separately makes using a smaller number of them preferable. While readability within the files may be worsened by this approach, some of the class definitions are too short to motivate 17 header files and 17 source files. Therefore, `piece` classes are divided between two pairs of files, one for the abstract classes and one for the non-abstract ones.

## 2.2 Functionality

The program incorporates all the mechanics of the standard Blokus game and allows multiple games to be played without being restarted. Moves of the players are stored simply by displaying the board and players' sets. The structure of the game is such that this unambiguously leads to a list of pieces used and positions at which they were placed. Since all the pieces "move" in the same way, by being placed on chosen board squares,

no list of allowed moves is provided. Each player can, however, look for all the available board positions a chosen piece in its current arrangement can occupy. This can be done at any moment during the game, apart from the first round. The `find_available_moves board` class member function is responsible for this mechanism. The function is also the central component of the `check_if_can_place` function, which runs automatically in order to check if the player has any moves left. The `find_available_moves` function, similarly to the `take_turn` function, works by calling the `place_piece` function. The source of the call is distinguished by two `bool` parameters, `test` and `players_test`. The `place_piece` function is responsible for making all the checks required to decide whether the chosen `piece` can be placed at the chosen `board` position. The functionality of the `take_turn` function is much wider than placing pieces on the board. It handles users' input, calls the `piece` member functions `rotate` and `flip` through instances of the `players_set` class and `board` member functions `find_available_moves` and `place_first_piece`. It also incorporates recursion. The `place_first_piece` function is necessary, as the mechanics of the first round are different to all the rounds that follow.

```
void pentomino::rotate() // used by 3 out of 4 classes derived from pentomino
{
    bool changed{ false }; // used for updating handle
    size_t divisor{ length };
    std::vector<std::pair<size_t, bool>> before; // piece before rotation
    // reserve space on the vector, it makes the program faster
    before.reserve(length * width);
    for (size_t i{}; i < length * width; i++) {
        before.push_back(std::make_pair(i, is_there_square[i]));
    }
    std::vector<std::pair<size_t, bool>> after(before.size()); // after rotation
    for (size_t i{}; i < length; i++) {
        std::copy_if(before.begin(), before.end(), after.begin() + i * width,
            [i, divisor](std::pair<size_t, bool> pair) {
            return (pair.first % divisor == divisor - 1 - i);
        }); // rotate by adding bools to the after vector in specific order
    }
    // the after vector is reversed compared to what is needed
    for (size_t i{}; i < length * width; i++) {
        is_there_square[i] = after.at(length * width - 1 - i).second;
        if (after.at(length * width - 1 - i).first == handle && !changed) {
            handle = i; // update handle
            changed = true; // avoid changing handle twice
        }
    }
    std::swap(length, width);
}
```

Figure 5: The `rotate` function presented here is a virtual function and a member of the `pentomino` class. The only class derived from the `pentomino` class which overwrites it is the `pentomino_51` class. Rotation consists of rearranging the order of the squares making up the `piece` within the rectangle and changing the rectangle position from vertical to horizontal or vice-versa. In order to obtain the correct order, the standard library `copy_if` function and a lambda capture are used in a loop, and then the order is reversed. The function also makes use of `std::pair` container, which is frequently used across the program.

The `piece` mechanics allow for repositioning of the `piece`'s before placing them on the

`board`. The functions responsible for that extensively use the standard library and lambda captures. The `rotate` function definition is displayed in Figure 5. The classes derived from the `piece` class whose instances do not fill out their entire rectangle with the squares contain static constant member `bool` vectors with arrangement information on the allowed square configurations. The vectors are used for `piece` objects construction in the `players_set` constructor. Since the vectors are made private, public static `get_construction_vector` functions are also implemented. One of them can be seen in Fiugre 6. Static vectors are also used for construction of the `board` object and in some of its functions.

```
static std::vector<bool> get_construction_vector(const char letter);
```

```
std::vector<bool> tetromino_32::get_construction_vector(const char letter)
{
    if (letter != 'L' && letter != 'T' && letter != 'Z') {
        throw("Exception: invalid letter.");
    }
    std::vector<bool> construction_vector;
    if (letter == 'L') {
        construction_vector = tetromino_L;
    }
    else if (letter == 'T') {
        construction_vector = tetromino_T;
    }
    else if (letter == 'Z') {
        construction_vector = tetromino_Z;
    }
    return construction_vector;
}
```

Figure 6: The declaration and definition of a static `get_construction_vector` function in the top and bottom panels, respectively. The function includes throwing an exception in case of an invalid input. Such practice is common in the program, however, there are no try or `catch` statements. In some cases, there is simply no need for them, as parameters are initialized without users' involvement. In places where there is users' input other methods of handling it are chosen.

The user interface of the program is constructed through numerous member functions of all the classes. All of them use the `windows.h` library in one or two ways. It is either explicitly used in order to change the colour of the console or calls one of graphics functions. The graphics functions are located in a separate pair of header and source files, whose relevant parts can be seen in Figure 7. All the pieces and the board are made out of squares created by `draw_square` function. In order to distinguish between two types of squares the program uses, namespaces are introduced. The `windows.h` library is also used in `main` in order to change the console to full screen, which is neccessary in order to fit all the components of the game.

```cpp
// allows moving around the console
void go_to_console_position(size_t x, size_t y);

// distinguishes two types of squares used in order to display the game
namespace top
{
    void draw_square(size_t x, size_t y);
}
namespace bottom
{
    void draw_square(size_t x, size_t y);
}
```

```cpp
void go_to_console_position(size_t x, size_t y)
{
    COORD position{ static_cast<SHORT>(x), static_cast<SHORT>(y) };
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), position);
}

void top::draw_square(size_t x, size_t y)
{
    go_to_console_position(x, y);
    std::cout << "    ";
    go_to_console_position(x, y + 1);
    std::cout << "____"; // separates the square from the square below
}

void bottom::draw_square(size_t x, size_t y)
{
    go_to_console_position(x, y);
    std::cout << "    ";
    go_to_console_position(x, y + 1);
    std::cout << "    "; // there is no square below, so no need to separate
}
```

Figure 7: The declarations and definitions of the graphics functions in the top and bottom panels, respectively.

# 3 Results

The output of the program is presented in Figure 8. The board is displayed on the left-hand side, the player's set in the bottom right corner. The summarized rules are located below the board. The user input panel and the option menu is in the top right corner. Each turn is taken in three stages. The question corresponding to the current stage is displayed in capital letters, and the cursor is placed next to it. The pieces are placed on the board by choosing the board square that the piece square marked with the piece number will be placed at.

Only specific inputs are allowed, the program accounts for the invalid ones. In theory, a player can type in an infinite number of commands every round, as there is no limit on number of rotations, flips or available moves searches. Players are also given an option to go back to the first stage and choose a different piece at any point during their turn by pressing b.

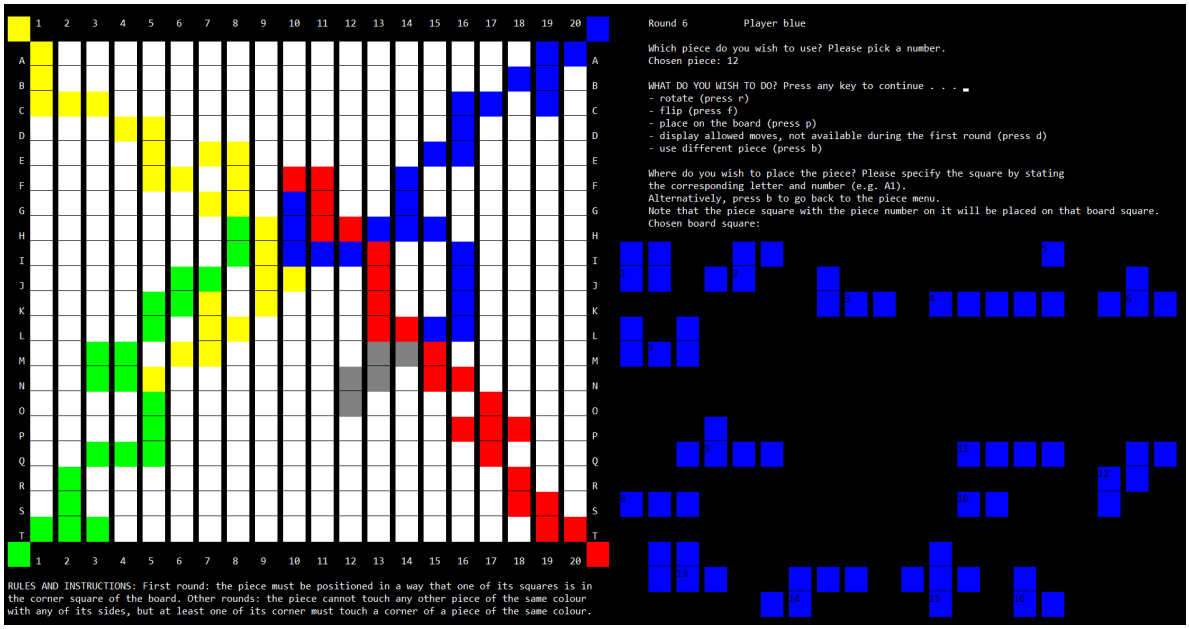


Figure 8: The board with a few pieces already on it. The pieces placed on the board disappear from the player's set panel. During any player's turn, only their pieces are displayed. The blue player has chosen the piece marked with number 12 and has pressed d. All the available board positions for that piece in the current arrangement are displayed one by one. The program waits for the player to press a key before proceeding to the next one.

# 4 Conclusions

Overall, the program functions well and has a user-friendly interface. The achieved functionality is beyond the expected minimum, and so is the class design. Various advanced features are used, improving the design of the program. The size of the board and the form of the pieces made it quite challenging to implement an efficient and clear method of associating a piece with a position on the board. The found method accomplishes its tasks well, however, it may not be obvious to a user who has not played before. While this aspect would be very difficult to improve, there are some that could be. The `handle` mechanism of some `piece` classes only works correctly for specific values. This does not cause any issues, as all the `piece`'s are constructed without users' involvement, however, it could turn out to be a limitation if the game was developed further. Additionally, the mechanisms responsible for displaying the game on the console could be made more flexible. Finally, an algorithm could be created in order to allow a smaller number of players to play while pieces of missing players are placed on the board by the program.

# References

[1] *Polyomino*, available at `https://en.wikipedia.org/wiki/Polyomino#Enumeration_of_polyominoes`, accessed on 2020/05/09.

[2] *Blokus*, available at `https://en.wikipedia.org/wiki/Blokus`, accessed on 2020/05/09.