Intermediate Programming

Day 34

Outline

- Exercise 33
- Object oriented design and Unified Modeling Language
- Review questions
- Final project

Exercise 33

Add a virtual **toString** function to `Aclass.h`

Aclass.h

```
...
class A
{
private:
	int a;
protected:
	double d;
	...
	virtual std::string toString( void ) const
	{
		std::stringstream sstream;
		sstream << "[Aclass: a = " << a << ", d = " << d;
		sstream << ", size = " << sizeof( A ) << "]";
		return sstream.str();
	}
	...
};
...
```

Exercise 33

Override `toString` in `Bclass.h`

Bclass.h

```
...
class B : public A
{
private:
	int b;
public:
	...
	std::string toString( void ) const override
	{
		std::stringstream sstream;
		sstream << "[Bclass: a = " << geta() << ", b = " << b <<  ", d = " << d;
		sstream << ", size = " << sizeof( B ) << "]";
		return sstream.str();
	}
};
```

Aclass.h

```
...
class A
{
private:
	int a;
protected:
	int geta( void ) const { return a; }
	...
};
...
```

Exercise 33

Add a pure **virtual** function **fun** to class ***A*** and implement it for class **B**

Aclass.h

```
...
class A
{
        ...
protected:
        ...
        virtual int fun( void ) const = 0;
        ...
};
...
```

Bclass.h

```
...
class B
{
        ...
protected:
        ...
        int fun( void ) const override { return geta() * b * d; }
        ...
};
...
```

Exercise 33

Create a class ***C***

Cclass.h

```
...
class C : public A
{
private:
	int e;
public:
	C( int val=0 ) : e(val) {}
	void sete( int val ) { e = val; }
	int fun( void ) const override { return e * geta() * d; }
	std::string toString( void ) const override
	{
		std::stringstream sstream;
		sstream << "[Cclass: a = " << ", d = " << d << ", e = " << e;
		sstream << ", size = " << sizeof( C ) << "]";
		return sstream.str();
	}
};
...
```

Outline

- Exercise 33
- Object oriented design and Unified Modeling Language
- Review questions
- Final project

OO Design & UML

In our code, the different classes can interact with each other:

- **Inheritance**
  A derived class can inherit from a base class
- **Aggregation**
  A class can contain a pointer/reference to another class as one of its members
- **Composition**
  A class can contain an object of another class as one of its members

A UML diagram can help us track the classes and the relationships between them.*

*In this lecture we will only be talking about a small subset of UML diagrams.

OO Design & UML

Classes:

- Typically represented by a rectangle with the class name

Visualization:

- Class: named rectangle

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { … }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

OO Design & UML

Inheritance:

- Represents an “is a” relationship
  - A Circle is a Shape
  - A Square is a Shape
  - A ShapeList is a Shape
- Typically represented as a (hollow) arrow from the derived class to the base

Visualization:

- Class: named rectangle
- Inheritance: (hollow) arrow from derived to base

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { … }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

OO Design & UML

Aggregation:

- Represents a “has a” relationship
  - A ShapeList has a Shape(s)
- Aggregated data **can** exist without the containing class
- Typically represented as a (hollow) diamond from the class being contained to the class containing

Visualization:

- Class: named rectangle
- Inheritance: (hollow) arrow from derived to base
- Aggregation: (hollow) diamond arrow to class with reference/pointer

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { … }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

OO Design & UML

Composition:

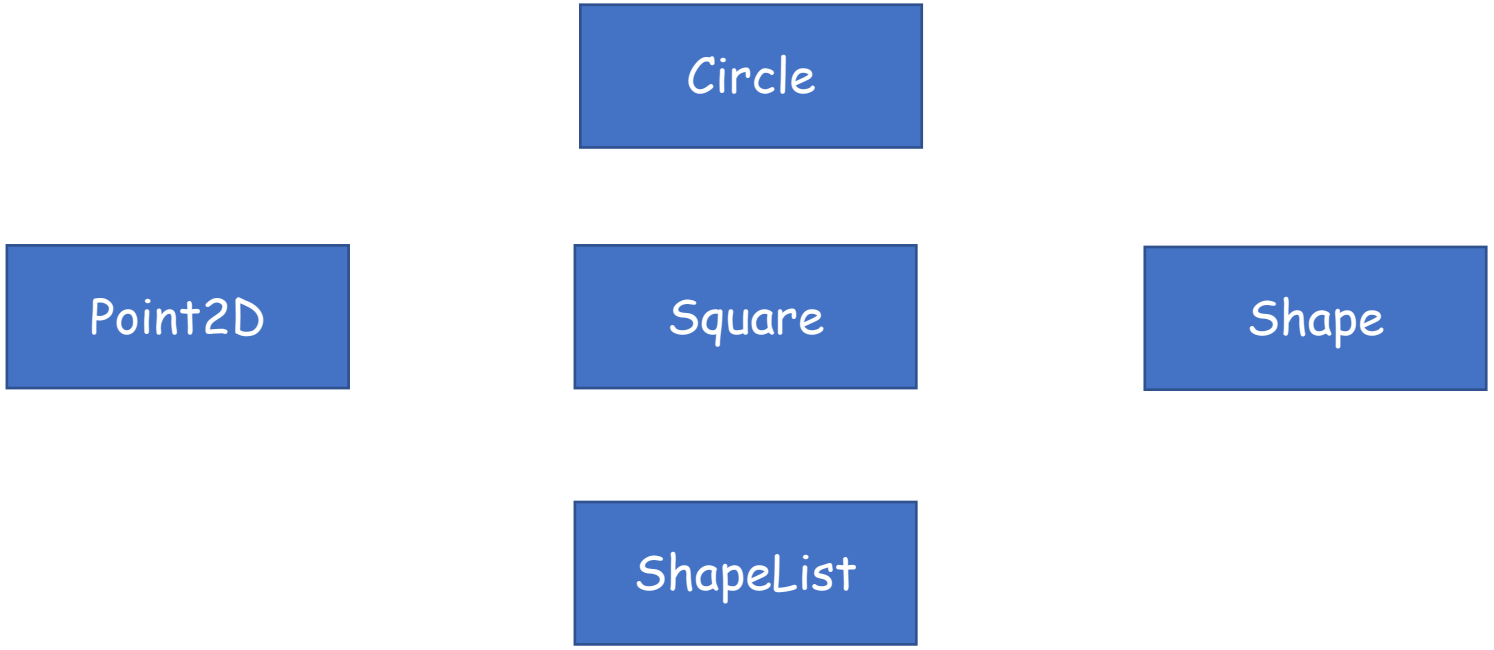
- Represents a “has a” relationship
  - A *Circle* has a *Point2D*
  - A *Square* has a *Point2D*
- Compositional data **cannot** exist without the containing class
- Typically represented as a (solid) diamond from the class contained to the class containing

Visualization:

- Class: named rectangle
- Inheritance: (hollow) arrow from derived to base
- Aggregation: (hollow) diamond arrow to class with reference/pointer
- Composition: (solid) diamond arrow to class containing object

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { … }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

OO Design & UML

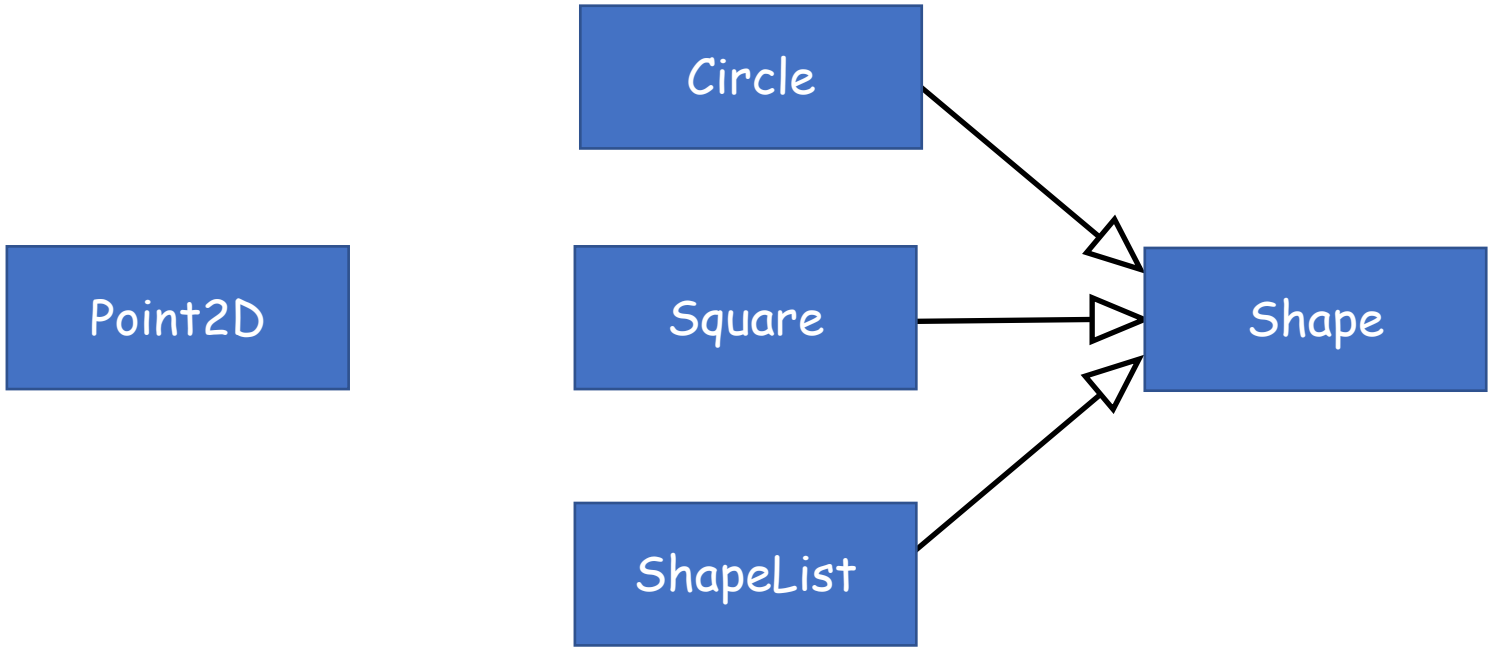


Visualization:

- **Class: named rectangle**
- Inheritance: (hollow) arrow from derived to base
- Aggregation: (hollow) diamond arrow to class with reference/pointer
- Composition: (solid) diamond arrow to class containing object

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { ... }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

OO Design & UML

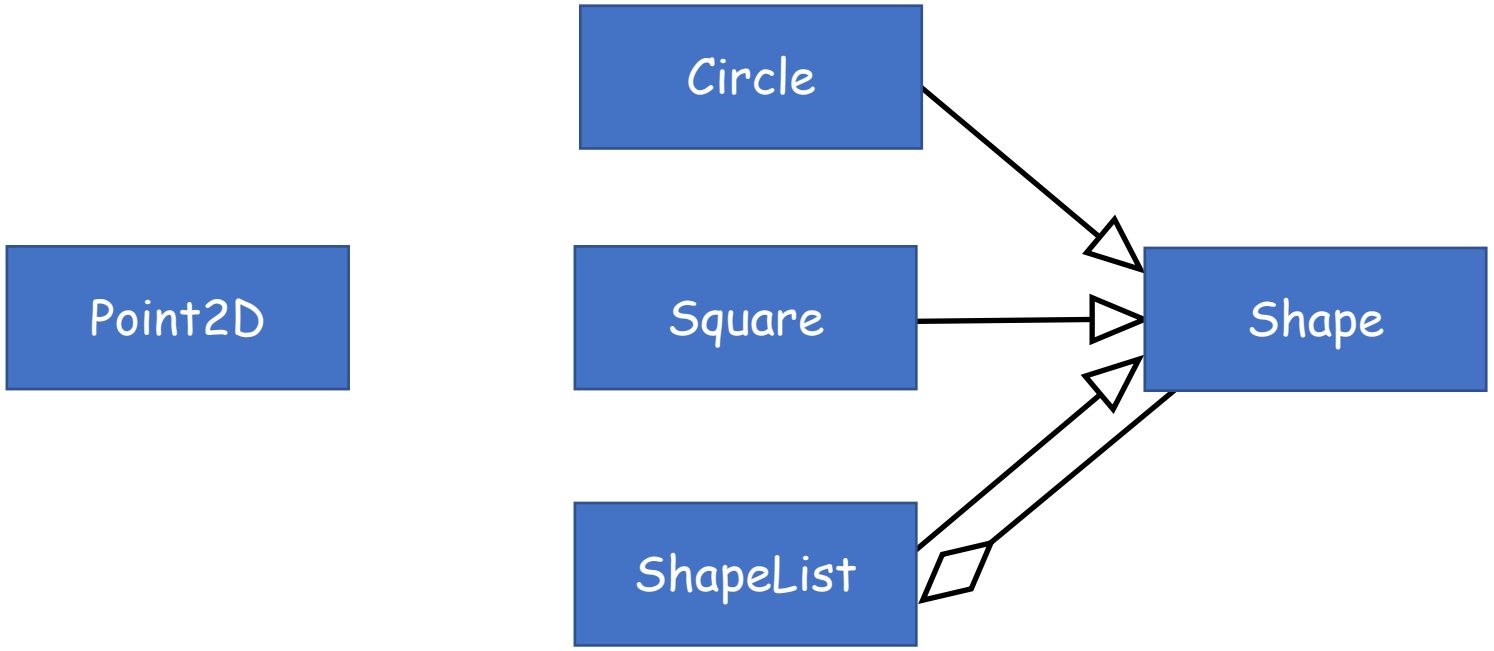


Visualization:

- Class: named rectangle
- **Inheritance: (hollow) arrow from derived to base**
- Aggregation: (hollow) diamond arrow to class with reference/pointer
- Composition: (solid) diamond arrow to class containing object

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { ... }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

OO Design & UML

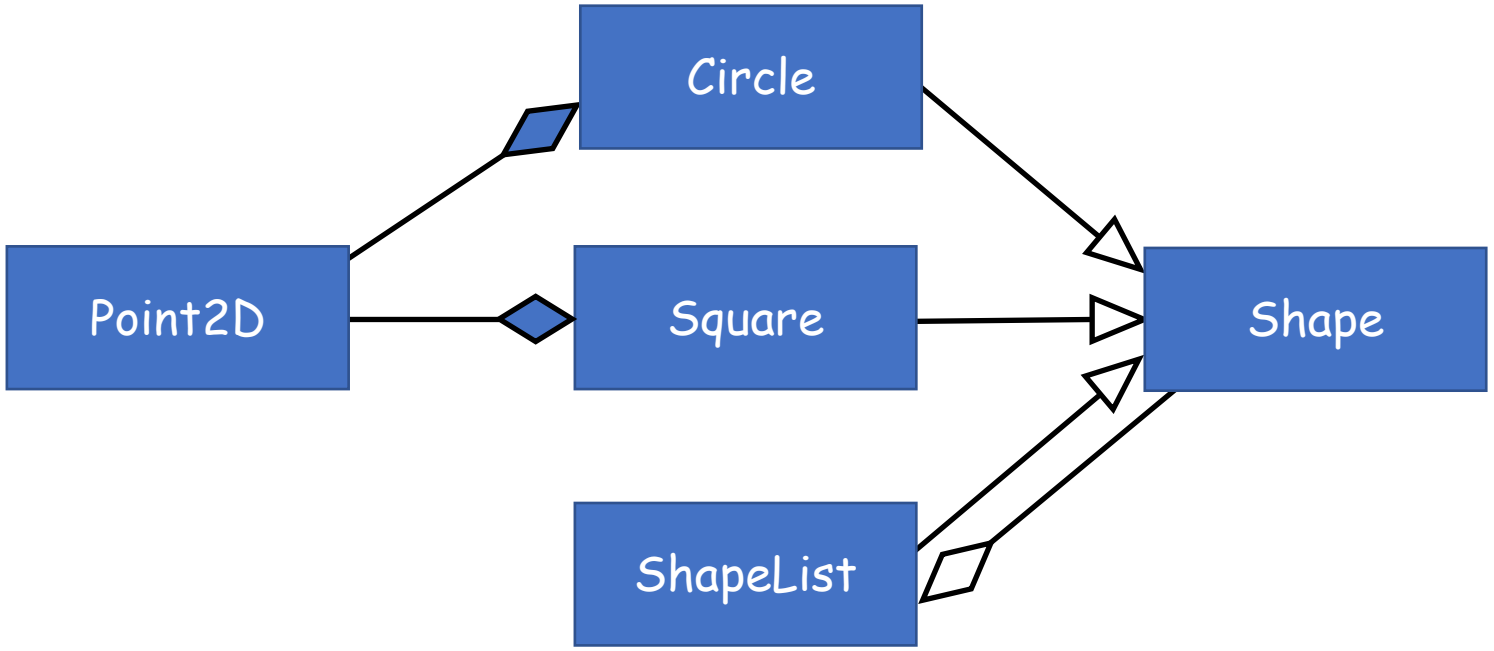


Visualization:

- Class: named rectangle
- Inheritance: (hollow) arrow from derived to base
- **Aggregation: (hollow) diamond arrow to class with reference/pointer**
- Composition: (solid) diamond arrow to class containing object

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { ... }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

OO Design & UML



Visualization:

- Class: named rectangle
- Inheritance: (hollow) arrow from derived to base
- Aggregation: (hollow) diamond arrow to class with reference/pointer
- **Composition: (solid) diamond arrow to class containing object**

```
class Point2D
{
public:
        double x , y;
};
class Shape
{
public:
        virtual double getArea( void ) const = 0;
        virtual void draw( void ) const = 0;
};
class Circle : public Shape
{
        Point2D p ; double r;
public:
        double getArea( void )  const { ... }
        void draw( void ) const { ... }
};
class Square : public Shape
{
        Point2D bottomLeft , topRight;
public:
        double getArea( void ) const { ... }
        void draw( void ) const { ... }
};
class ShapeList : public Shape
{
        std::vector< Shape * > shapes;
public:
        void getArea( void ) const { ... }
        void draw( void ) const { ... }
};
```

Outline

- Exercise 33
- Object oriented design and Unified Modeling Language
- Review questions
- Final project

Review questions

1. What is UML?

Unified Modeling Language - way to visually represent class diagrams and other software engineering components

Review questions

2. What type of class relationship is likely to exist between a class that represents `Bathroom` objects and one that represents `Apartment` objects?

An `Apartment` "has a" `Bathroom`

Review questions

3. What type of class relationship is likely to exist between a class that represents **Apartment** objects and one that represents **Housing** objects?

An **Apartment** "is a" **Housing**

Review questions

4. Which of Bathroom, Apartment, Housing would likely be an abstract class?

Housing since it is not object specific but represents a general type instead

Outline

- Exercise 12-1
- Object oriented design and Unified Modeling Language
- Review questions
- Final project

Final project

Chess:

- Two players
- 8x8 tiled board
- Each player starts with 16 pieces
  - 2 rooks
  - 2 knights
  - 2 bishops,
  - 1 King
  - 1 Queen
  - 8 pawns

Final project

Turn:

- Players alternate turns
- The player whose turn it is moves one of their pieces:
  - Move:
    The piece goes from the tile it's on to an empty tile
  - Capture:
    The piece goes from the tile it's on to a tile with an opponent's piece, and the opponent's piece is removed from the game
- For most pieces valid move and capture shapes are the same

Final project

Note:
1. Piece is an abstract class because legal_move_shape is pure virtual.
2. By default legal_capture_shape just checks if the move shape is legal.

- The player whose turn it is mov
  - Move:
    The piece goes from the tile it's
  - Capture:
    The piece goes from the tile it's
    and the opponent's piece is rem
- For most pieces valid move and

Piece.h

```
…
namespace Chess
{
        class Piece
        {
        public:
                …
                bool is_white() const { … }
                virtual bool legal_move_shape( … ) const = 0;
                virtual bool legal_capture_shape( … ) const
                { return legal_move_shape( … ); }
                virtual char to_ascii() const = 0;
                …
        };
}
```

King.h

```
namespace Chess
{
        class King : public Piece
        {
                ...
        };
}
```

Bishop.h

```
namespace Chess
{
        class Bishop : public Piece
        {
                ...
        };
}
```

Rook.h

```
namespace Chess
{
        class Rook : public Piece
        {
                ...
        };
}
```

Queen.h

```
namespace Chess
{
        class Queen : public Piece
        {
                ...
        };
}
```

Knight.h

```
namespace Chess
{
        class Knight : public Piece
        {
                ...
        };
}
```

Pawn.h

```
namespace Chess
{
        class Pawn : public Piece
        {
                ...
        };
}
```

Note:

1. Piece is a
2. By defaul

se legal_
virtual.
pe just ch
e is lega

Queen, King, Bishop, Knight, Rook, and Pawn all derive from Piece.

and the opponent's piece is rem

- For most pieces moving and ca

Piece.h

```
...
namespace Chess
{
        class Piece
        {
        public:
                ...
                bool is_white() const { ... }
                virtual bool legal_move_shape( ... ) const = 0;
                virtual bool legal_capture_shape( ... ) const
                { return legal_move_shape( ... ); }
                virtual char to_ascii() const = 0;
                ...
        };
}
```

King.h

```
namespace Chess
{
        class King : public Piece
        {
                ...
        };
}
```

Bishop.h

```
namespace Chess
{
        class Bishop : public Piece
        {
                ...
        };
}
```

Rook.h

```
namespace Chess
{
        class Rook : public Piece
        {
                ...
        };
}
```

Queen.h

```
namespace Chess
{
        class Queen : public Piece
        {
                ...
        };
}
```

Knight.h

```
namespace Chess
{
        class Knight : public Piece
        {
                ...
        };
}
```

Pawn.h

```
namespace Chess
{
        class Pawn : public Piece
        {
                ...
        };
}
```

Note:

1. Piece is a
2. By defaul

se legal_
virtual.
pe just ch
e is lega

Queen, King, Bishop, Knight, Rook, and Pawn all derive from Piece.

For each of the derived classes, you will define the legal_move_shape member function.

If it needs it, you will also override the legal_capture_shape member function.

Piece.h

```
...
namespace Chess
{
        class Piece
        {
        public:
                ...
                bool is_white() const { ... }
                virtual bool legal_move_shape( ... ) const = 0;
                virtual bool legal_capture_shape( ... ) const
                { return legal_move_shape( ... ); }
                virtual char to_ascii() const = 0;
                ...
        };
}
```

Final project

At each turn:

- Identify whether checkmate has happened
- Identify whether a player is in check
- Identify whether stalemate has happened
- Query the player until they provide legal move/capture (or they quit)

Final project

You will define the:

- in_mate,
- in_check,
- in_stalemate, and
- make_move

member functions for the Game class.

Note:
The main function does not switch the players. You do that once a successful move has been made (in make_move).

main.cpp

```
...

int main( int argc , char* argv[] )
{
        ...
        while(  !game_over )
        {
                ...
                game.get_board().display();
                ...
                if ( game.turn_white() )  std::cout << "White's move." << std::endl;
                else                      std::cout << "Black's move." << std::endl;
                ...
                if      ( game.in_mate( game.turn_white() ) ) { ... }
                else if( game.in_check( game.turn_white() ) ) { ... }
                else if( game.in_stalemate( game.turn_white() ) ) { ... }
                ...
                game.make_move( ... );
        }
}
```

Final project

in_check:

A player is in check if:

- It's the player's turn
- The player's king is under attack ⇔ There's a legal capture move the opponent can make that would take the player's king
- There is a legal move/capture the player can do that would make the king not be under under attack

⇒ If a player is in check, they have to move/capture to get out of it.

Final project

in_mate:

A player is in checkmate if:

- It's the player's turn
- The player's king is under attack ⇔ There's a legal capture move the opponent can make that would take the player's king
- There is no legal move/capture the player can do that would make the king not be under attack

⇒ If a player is in mate, they lose.

Final project

in_stalemate:

A player is in stalemate if:

- It's the player's turn
- The player's king is not under attack
- There is no legal move/capture the player can do that would make the king not be under attack

⇒ If a player is in mate, it's a tie.

Final project

make_move:

A move is legal if:

- The player moves their own piece
- It has a legal move shape (if there is no piece is at the endpoint)
- It has a legal capture shape (if there is an opponent's piece is at the endpoint)
- It does not pass over other pieces (if it moves horizontally, vertically, or diagonally)
- It does not expose the player's king to attack

Final project

make_move:

Hint:
- ✓ You have already implemented the `in_check` member function.
- ✗ You don't want to make the move and invoke the `in_check` member function, because if the move does put the player in check, you will need to "unwind" it.
- ⇒ Make a copy of the `Board`, make the move on the copy, and check if the move puts you in check there.

  - It does not expose the player's king to attack

Final project

make_move:

Hint:

- ✓ You have already implemented the **in_check** member function.
- ✗ You don't want to make the move and invoke the **in_check** member function, because if the move does put the player in check, you will need to "unwind" it.

⇒ Make a copy of the **Board**, make the move on the copy, and check if the move puts you in check there.

Note:

The **make_move** member function will try to make the move. If the move is not legal, it will throw an exception. It is your responsibility to manage the exception handling.

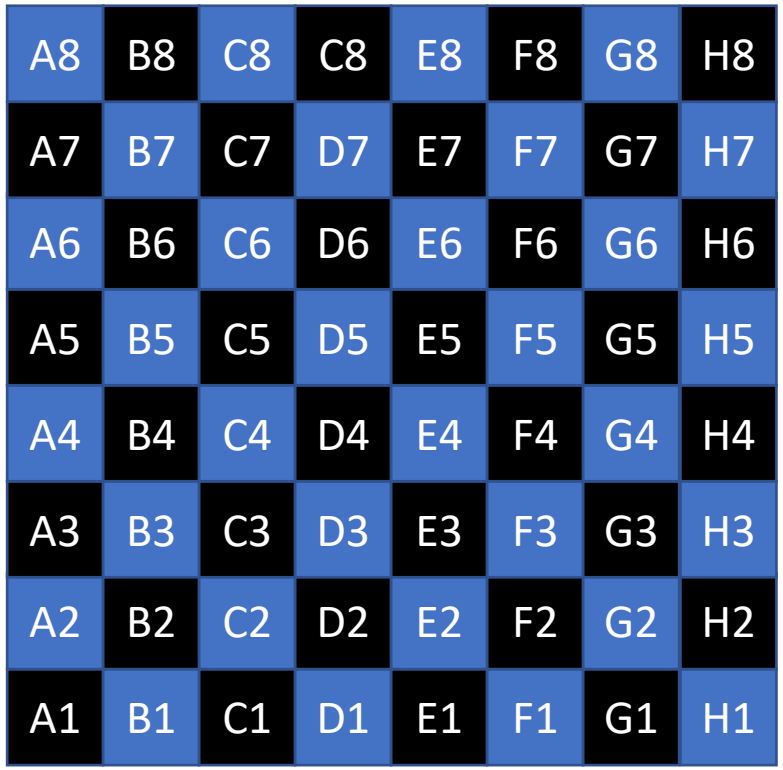
Final project

Representation of a position:

A position on the board is indexed by a pair of values:

- The first is a letter in the range {‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’} (all caps) specifying the column.
- The second is a number in the range {‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’} specifying the row

Note:
In the game, a position is represented by a `std::pair< char , char >`.



Final project

Board.h

```
...
namespace Chess
{
	class Board
	{
		...
	private:
		std::map< Position , Piece * > occ;
	};
}
```

Representation of the games state:

The **Board** class stores the game state.

The state is represented as a **std::map** whose keys are positions, and whose values are **Piece** pointers.

Final project

Board.h

```
…
namespace Chess
{
        class Board
        {
                …
        private:
                std::map< Position , Piece * > occ;
        };
}
```

Representation of the games state:

You will define the operator:

```
const Piece* operator() ( const Position &position ) const;
```

This returns a pointer to the Piece at the prescribed position, if there is a piece there.

Otherwise it returns a nullptr.

Final project

Board.h

```
...
namespace Chess
{
        class Board
        {
                ...
        private:
                std::map< Position , Piece * > occ;
        };
}
```

Representation of the games state:

You will define the member function:

```
bool add_piece( const Position &position , const char &piece_designator );
```

This tries to add a derived `Piece` of type specified by `piece_designator` to the board.

It returns `false` if either the position is off the board or there is already a `Piece` at the prescribed position.

It returns `true` if the derived `Piece` was successfully added.

Final project

Representation of the games state:

You will define the member function:

```
bool add_piece( const Position &position , const char &piece_designator );
```

The piece_designator is a char:

- 'K'/'k': king
- 'Q'/'q': queen
- 'B'/'b': bishop
- 'N'/'n': knight
- 'R'/'r': rook
- 'P'/'p': pawn
- 'M'/'m': mystery

Upper-case is white and lower-case is black

Board.h

```
...
namespace Chess
{
	class Board
	{
		...
	private:
		std::map< Position , Piece * > occ;
	};
}
```

Piece.h

```
namespace Chess
{
	class Piece
	{
	public:
		bool is_white() const { ... }
		virtual bool legal_move_shape( ... ) const = 0;
		virtual bool legal_capture_shape( ... ) const
		{ return legal_move_shape( ... ); }
		virtual char to_ascii() const = 0;
		...
	};
}
```

Final project

Representation of the games state:

You will define the member function:

```
void display() const;
```

Draws the board to **std::cout**.

Board.h

```
…
namespace Chess
{
        class Board
        {
                …
        private:
                std::map< Position , Piece * > occ;
        };
}
```

Final project

Board.h

```
…
namespace Chess
{
        class Board
        {
                …
        private:
                std::map< Position , Piece * > occ;
        };
}
```

Representation of the games state:

You will define the member function:

```
bool has_valid_kings() const;
```

Checks that there is exactly one white King and one black King on the board.

Final project

The **Mystery** class:

Assuming you have implemented your code correctly, we should be able to introduce our own piece, with its own legal_move_shape member function (and possibly legal_capture_shape), and play it within your chess game.

Mystery.h

```
…
namespace Chess
{
	class Mystery : public Piece
	{
	public:
		bool legal_move_shape( const Position &start , const Position &end ) const;
		…
	};
}
```