600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Anouncements

- The final project web-page and code have been modified.
  - Be sure to get the latest skeleton.

Outline

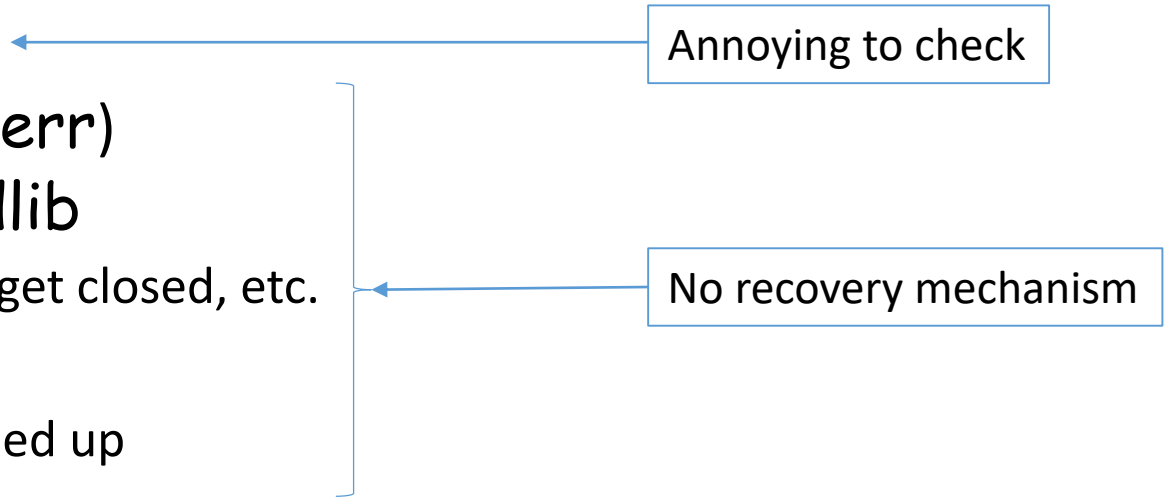
- Exceptions
- Final Project

Exceptions

- Things can go wrong at run-time:
  - Invalid data, file I/O problems, arithmetic operation problems, ...
- How should we deal with run-time error conditions?
  - Have a function return an error code
  - Display error messages (often using `cerr`)
  - Bail out using `exit( int )` - need `cstdlib`
    - "clean exit": destructors get called, files get closed, etc.
  - Bail out using `abort( void )`
    - "hard exit": nothings gets called or cleaned up

Exceptions

- Things can go wrong at run-time:
  - Invalid data, file I/O problems, arithmetic operation problems, ...
- How should we deal with run-time error conditions?
  - Have a function return an error code
  - Display error messages (often using `cerr`)
  - Bail out using `exit( int )` - need `cstdlib`
    - "clean exit": destructors get called, files get closed, etc.
  - Bail out using `abort( void )`
    - "hard exit": nothings gets called or cleaned up



Exceptions

- Exceptions are objects that help us manage run-time error situations
  - The class `std::exception` is a type in the standard library
  - But we can define our own exception classes too

- We may employ `throw` statements when we identify an error situation that we don't want to handle immediately (or at all)...

```
throw std::exception();
```

- ... and `try` / `catch` blocks to indicate situations we'd like to handle, and how to handle them (whether we threw the associated exceptions ourselves or not)

Exceptions

- Exceptions are objects that hel
  - The class **std::exception** is a ty
  - But we can define our own exce
- We may employ **throw** statem
situation that we don't want to

throw st

- ... and **try** / **catch** blocks to indicate situa dle, and how to handle them (whether we threw the associated exceptions ourselves or not)

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
	try
	{
		throw std::exception();
	}
	catch( std::exception& ex )
	{
		std::cerr << "Exception: " << ex.what() << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
Exception: std::exception
>>
```

Exceptions

- The `std::exception` class has a virtual method that returns a string describing the exception (and will not to throw an exception):

```
const char* what( void ) const noexcept;
```

- We can define our own exception class that we can `throw` / `catch`
  - Although we don't have to, we should make it derive from `std::exception`
  - We can over-ride the `exception::what` method

Exceptions

- The **std::exception** class has a
  describing the exception (and w

  const char* what(

- We can define our own excepti
  - Although we don't have to, we sh
  - We can over-ride the **exception**

main.cpp

```
#include <iostream>
#include <exception>

class MyException : public std::exception
{
public:
	const char* what( void ) const noexcept
	{
		return "my exception";
	}
};
int main( void )
{
	try
	{
		throw MyException();
	}
	catch( std::exception& ex )
	{
		std::cerr << "Exception: " << ex.what() << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
Exception:  my exception
>>
```

try/catch blocks

- If no exception is thrown by code in try block, then catch block(s) are all skipped, and execution continues normally.

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
	try
	{
		std::cout << "trying" << std::endl;
	}
	catch( std::exception& )
	{
		std::cerr << "catching" << std::endl;
	}
	std::cout << "done" << std::endl;
	return 0;
}
```

```
>> ./a.out
trying
done
>>
```

try/catch blocks

- If an exception is thrown by code in try block, execution immediately jumps to the first matching catch block (if one exists), and code in that single catch block is executed, then execution continues normally after catch block.
  - A catch block "matches" if the type of the exception is derived from the parameter type

main.cpp

```
#include <iostream>
#include <exception>
class MyException : public std::exception
{
};

int main( void )
{
	try{ throw MyException(); }
	catch( MyException& )
	{
		std::cerr << "caught mine" << std::endl;
	}
	catch( std::exception& )
	{
		std::cerr << "caught generic" << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
caught mine
>>
```

try/catch blocks

- If an exception is thrown by code in try block, execution immediately jumps to the first matching catch block (if one exists), and code in that single catch block is executed, then execution continues normally after catch block.
  - A catch block "matches" if the type of the exception is derived from the parameter type
  - List the catch blocks in order from most derived to least!

main.cpp

```
#include <iostream>
#include <exception>
class MyException : public std::exception
{
};

int main( void )
{
	try{ throw MyException(); }
	catch( std::exception& )
	{
		std::cerr << "caught generic" << std::endl;
	}
	catch( MyException& )
	{
		std::cerr << "caught mine" << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
caught generic
>>
```

try/catch blocks

- If an exception is thrown by code in try block, but no suitable catch block exists, the exception is passed up the call stack

main.cpp

```
#include <iostream>
#include <exception>

void foo( void ){ throw std::exception(); }

int main( void )
{
	try{ foo(); }
	catch( std::exception& )
	{
		std::cerr << "caught generic" << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
caught generic
>>
```

try/catch blocks

- If an exception is thrown by code in try block, but no suitable catch block exists, the exception is passed up the call stack
  - If the exception isn't caught, the code terminates

main.cpp

```
#include <iostream>
#include <exception>

void foo( void ){ throw std::exception(); }

int main( void )
{
      std:: cout << "pre foo" << std:: endl;
      foo();
      std::cout << "post foo" << std::endl;
      return 0;
```

```
>> ./a.out
pre foo
terminate called after throwing an instance of 'std::exception'
  what():  std::exception
Abort (core dumped)
>>
```

main.cpp

```
#include <iostream>
#include <exception>
using namespace std;

int main( void )
{
	try
	{
		cout << "a" << endl;
		throw std::exception();
		cout << "b" << endl;
	}
	catch( exception& ){ cout << "caught exception!" << endl; }
	return 0;
}
```

```
>> ./a.out
a
caught exception
>>
```

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

int main( void )
{
	try{ throw overflow_error( "ran out of space!" ); }
	catch( invalid_argument &e ){ cout << "got invalid argument: e.what() = "     << e.what() << endl; }
	catch( overflow_error   &e ){ cout << "got overflow exception: e.what() = " << e.what() << endl; }
	catch( exception        &e ){ cout << "got base exception: e.what() = "      << e.what() << endl; }
	return 0;
}
```

```
>> ./a.out
got overflow exception: e.what()=ran out of space!
>>
```

Note: Most classes derived from **std::exception** define a constructor that takes a string argument

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

int bar( int x )
{
	if( x>100000 ){ throw range_error( "x out of range ( " + to_string(x) + " > 100000 )" ); }
	return x * 10;
}

void foo( int &x ){ x = bar(x); }

int main( void )
{
	try
	{
		int num = 10;
		for( int i=0 ; i<10 ; i++ ){ foo( num ) ; cout << "(" << i << ") " << num << endl; }
	}
	catch( range_error &e ){ cout << " caught range error: " << e.what() << endl; }
	return 0;
}
```

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

int bar( int x )
{
	if( x>100000 ){ throw range_error( "x out of range ( " + to_string(x) + " > 100000 )" ); }
	return x * 10;
}

void foo( int &x ){ x = bar(x); }

int main( void )
{
	try
	{
		int nu
		for( i
	}
	catch( rang
	return 0;
}
```

```
>> ./a.out
(0) 100
(1) 1000
(2) 10000
(3) 100000
(4) 1000000
 caught range error: x out of range ( 1000000 > 100000 )
>>
```

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

int main( void )
{
	try
	{
		int *array = new int[100000000000];
		array[0] = 10;
	}
	catch( bad_alloc &bae ){ cout << "error while allocating: " << bae.what() << endl; }
	return 0;
}
```

```
>> ./a.out
error while allocating: std::bad_alloc
>>
```

Note: We are trying to allocate 4 terabytes of data. That's likely to exceed RAM.

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>
#include <vector>

using namespace std;

int main( void )
{
	try
	{
		std::vector<double> v(10);
		v.at(9) = 20;
		v.at(10) = 21;
	}
	catch( exception &ex ){ cout << "error with vector: " << ex.what() << endl; }
	return 0;
}
```

```
>> ./a.out
error with vector: vector::_M_range_check: __n (which is 10) >= this->size() (which is 10)
>>
```

Note: The vector::at method tests if the index is in bounds and throws an exception if it's not

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

class Boom : public exception
{
public:
	virtual const char* what() const noexcept { return "BOOM!"; }
};

int main( void )
{
	try
	{
		Boom firecracker;
		throw firecracker;
	}
	catch( exception &e ){ cout << "e.what: " << e.what() << endl; }
	catch( Boom &b ){ cout << "WILL NEVER GET HERE" << endl; }
	return 0;
```

```
>> ./a.out
e.what: BOOM!
>>
```

main.cpp

```
#include <iostream>
#include <exception>

using namespace std;

int main( void )
{
      try
      {
            throw 7;
      }
      catch( int &i ){ cout << "caught: " << i << endl; }
      return 0;
}
```

```
>> ./a.out
caught 7
>>
```

Outline

- Exceptions
- Final Project

Class / **enum** Layout

- Piece
  - Bishop
  - King
  - Knight
  - Pawn
  - Queen
  - Rook
- AbstractPieceFactory
  - PieceFactory< PieceType >
- Board
  - ChessGame
- Position
- Player
- Prompts
- Terminal

Outline

- Exceptions
- Final Project
  - Generic (abstract) classes
  - Chess classes

Piece: game.h

```
class Piece
{
public:
	virtual ~Piece( void ) {}
	Player owner( void ) const { return m_owner; }
	int id( void ) const { return m_id; }
	virtual int validMove( Position start , Position end , const Board& board ) const = 0;
protected:
	Player m_owner;
	int m_id;
	Piece( Player owner , int id ) : m_owner( owner ) , m_id( id ) {}
};
```

- An abstract class that describes the pieces in the game
  - Bishops, knights, etc. all derive from Piece

Piece: game.h

```
class Piece
{
public:
        virtual ~Piece( void ) {}
        Player owner( void ) const { return m_owner; }
        int id( void ) const { return m_id; }
        virtual int validMove( Position start , Position end , const Board& board ) const = 0;
protected:
        Player m_owner;
        int m_id;
        Piece( Player owner , int id ) : m_owner( owner ) , m_id( id ) {}
};
```

- An abstract class that describes the pieces in the game
- Every Piece must be able to check if a move is valid

Piece: game.h

```
class Piece
{
public:
        virtual ~Piece( void ) {}
        Player owner( void ) const { return m_owner; }
        int id( void ) const { return m_id; }
        virtual int validMove( Position start , Position end , const Board& board ) const = 0;
protected:
        Player m_owner;
        int m_id;
        Piece( Player owner , int id ) : m_owner( owner ) , m_id( id ) {}
};
```

- An abstract class that describes the pieces in the game
- Every Piece must be able to check if a move is valid
- A Piece has an integer identifier and is assigned an owner

Piece: game.h

```
class Piece
{
public:
        virtual ~Piece( void ) {}
        Player owner( void ) const { return m_owner; }
        int id( void ) const { return m_id; }
        virtual int validMove( Position start , Position end , const Board& board ) const = 0;
protected:
        Player m_owner;
        int m_id;
        Piece( Player owner , int id ) : m_owner( owner ) , m_id( id ) {}
};
```

- An abstract class that describes the pieces in the game
- Every Piece must be able to check if a move is valid
- A Piece has an integer identifier and is assigned an owner
- It has a protected constructor:

  ⇒ You cannot create a new piece directly

AbstractPieceFactory: game.h

```
class AbstractPieceFactory
{
public:
        virtual Piece* newPiece( Player owner ) const = 0;
        virtual ~AbstractPieceFactory( void ) {}
};
```

- An abstract class for synthesizing a particular type of Piece
  - There is a separate factory for each Piece

AbstractPieceFactory: game.h

```
class AbstractPieceFactory
{
public:
        virtual Piece* newPiece( Player owner ) const = 0;
        virtual ~AbstractPieceFactory( void ) {}
};
```

- An abstract class for synthesizing a particular type of Piece
- It takes in the owner and returns a pointer to a new (derived) Piece assigned to the specified player

PieceFactory: game.h

```
template< class PieceType >
class PieceFactory : public AbstractPieceFactory
{
public:
	PieceFactory( int id ) : m_id( id ) {}
	Piece* newPiece( Player owner ) const override { return new PieceType( owner , m_id ); }
protected:
	int m_id;
};
```

- A templated class that derives from AbstractPieceFactory
  - Though not enforced, the assumption is that PieceType is a class that is derived from Piece

PieceFactory: game.h

```
template< class PieceType >
class PieceFactory : public AbstractPieceFactory
{
public:
        PieceFactory( int id ) : m_id( id ) {}
        Piece* newPiece( Player owner ) const override { return new PieceType( owner , m_id ); }
protected:
        int m_id;
};
```

- A templated class that derives from AbstractPieceFactory
- Has a unique integer identifier associated with its (derived) Piece

PieceFactory: game.h

```
template< class PieceType >
class PieceFactory : public AbstractPieceFactory
{
public:
	PieceFactory( int id ) : m_id( id ) {}
	Piece* newPiece( Player owner ) const override { return new PieceType( owner , m_id ); }
protected:
	int m_id;
};
```

- A templated class that derives from AbstractPieceFactory
- Has a unique integer identifier associated with its (derived) Piece
- Can construct a new (derived) Piece with a prescribed owner
  - Assumes that PieceType has a constructor that takes an owner and identifier

Board: game.h

```
class Board
{
	using PieceGenMap = std::map< int , AbstractPieceFactory* >;
protected:
	PieceGenMap m_registeredFactories;
	unsigned int m_width , m_height;
	int m_turn;
	std::vector<Piece*> m_pieces;
	unsigned int index( Position position ) const { return position.y * m_width + position.x; }
	Piece* newPiece( int id , Player owner );
	bool addFactory( AbstractPieceFactory* );
};
```

- A square **width** x **height** grid tracking the players' turn

Board: game.h

```
class Board
{
	using PieceGenMap = std::map< int , AbstractPieceFactory* >;
protected:
	PieceGenMap m_registeredFactories;
	unsigned int m_width , m_height;
	int m_turn;
	std::vector<Piece*> m_pieces;
	unsigned int index( Position position ) const { return position.y * m_width + position.x; }
	Piece* newPiece( int id , Player owner );
	bool addFactory( AbstractPieceFactory* );
};
```

- A square **width** x **height** grid tracking the players' turn
- Each cell stores at most one piece

Board: game.h

```
class Board
{
	using PieceGenMap = std::map< int , AbstractPieceFactory* >;
protected:
	PieceGenMap m_registeredFactories;
	unsigned int m_width , m_height;
	int m_turn;
	std::vector<Piece*> m_pieces;
	unsigned int index( Position position ) const { return position.y * m_width + position.x; }
	Piece* newPiece( int id , Player owner );
	bool addFactory( AbstractPieceFactory* );
};
```

- A square **width** x **height** grid tracking the players' turn
- Each cell stores at most one piece
- A board stores factories for each **Piece** type

Board: game.h

```
class Board
{
	using PieceGenMap = std::map< int , AbstractPieceFactory* >;
protected:
	PieceGenMap m_registeredFactories;
	unsigned int m_width , m_height;
	int m_turn;
	std::vector<Piece*> m_pieces;
	unsigned int index( Position position ) const { return position.y * m_width + position.x; }
	Piece* newPiece( int id , Player owner );
	bool addFactory( AbstractPieceFactory* );
};
```

- A square **width** x **height** grid tracking the players' turn
- Each cell stores at most one piece
- A board stores factories for each **Piece** type
  - Has a method for adding a factory

Board: game.h

```
class Board
{
	using PieceGenMap = std::map< int , AbstractPieceFactory* >;
protected:
	PieceGenMap m_registeredFactories;
	unsigned int m_width , m_height;
	int m_turn;
	std::vector<Piece*> m_pieces;
	unsigned int index( Position position ) const { return position.y * m_width + position.x; }
	Piece* newPiece( int id , Player owner );
	bool addFactory( AbstractPieceFactory* );
};
```

- A square **width** x **height** grid tracking the players' turn
- Each cell stores at most one piece
- A board stores factories for each **Piece** type
  - Has a method for adding a factory
  - And a method for synthesizing a **Piece** with a particular id and owner

Board: game.h

```
class Board
{
public:
	Board( unsigned int w , unsigned int h , int turn=1 ) :
		m_width( w ) , m_height( h ) , m_turn( turn ) , m_pieces( w*h , nullptr ) {}
	virtual ~Board();
	unsigned int width( void ) const { return m_width; }
	unsigned int height( void ) const { return m_height; }
	bool initPiece( int id , Player owner , Position p );
	Piece* getPiece( Position p ) const;
	Player playerTurn( void ) const { return static_cast<Player>(!(m_turn % 2)); }
	int turn( void ) const { return m_turn; }
	bool validPosition( Position p ) const { return p.x < m_width && p.y < m_height; }
	virtual int makeMove( Position start , Position end ) = 0;
	virtual bool gameOver( void ) const = 0;
	virtual int run( void );
};
```

- Every board must be able to try and let a user move
  - Will return >=0 if the move is invalid

Board: game.h

```
class Board
{
public:
	Board( unsigned int w , unsigned int h , int turn=1 ) :
		m_width( w ) , m_height( h ) , m_turn( turn ) , m_pieces( w*h , nullptr ) {}
	virtual ~Board();
	unsigned int width( void ) const { return m_width; }
	unsigned int height( void ) const { return m_height; }
	bool initPiece( int id , Player owner , Position p );
	Piece* getPiece( Position p ) const;
	Player playerTurn( void ) const { return static_cast<Player>(!(m_turn % 2)); }
	int turn( void ) const { return m_turn; }
	bool validPosition( Position p ) const { return p.x < m_width && p.y < m_height; }
	virtual int makeMove( Position start , Position end ) = 0;
	virtual bool gameOver( void ) const = 0;
	virtual int run( void );
};
```

- Every board must be able to try and let a user move
- Every board must be able to determine if the game is over

Board: game.h

```
class Board
{
public:
	Board( unsigned int w , unsigned int h , int turn=1 ) :
		m_width( w ) , m_height( h ) , m_turn( turn ) , m_pieces( w*h , nullptr ) {}
	virtual ~Board();
	unsigned int width( void ) const { return m_width; }
	unsigned int height( void ) const { return m_height; }
	bool initPiece( int id , Player owner , Position p );
	Piece* getPiece( Position p ) const;
	Player playerTurn( void ) const { return static_cast<Player>(!(m_turn % 2)); }
	int turn( void ) const { return m_turn; }
	bool validPosition( Position p ) const { return p.x < m_width && p.y < m_height; }
	virtual int makeMove( Position start , Position end ) = 0;
	virtual bool gameOver( void ) const = 0;
	virtual int run( void );
};
```

- Every board must be able to try and let a user move
- Every board must be able to determine if the game is over
- Every board must be able to read prompts, try moves, switch players

Outline

- Exceptions
- Final Project
  - Generic (abstract) classes
  - Chess classes

Pawn…: chess.[h/cpp]

```
class Pawn : public Piece
{
protected:
	friend PieceFactory< Pawn >;
	Pawn( Player owner , int id ) : Piece( owner , id ) {}
public:
	int validMove( Position start , Position end , const Board& board ) const override;
};
```

- A derived Piece class for chess

Pawn…: chess.[h/cpp]

```
class Pawn : public Piece
{
protected:
	friend PieceFactory< Pawn >;
	Pawn( Player owner , int id ) : Piece( owner , id ) {}
public:
	int validMove( Position start , Position end , const Board& board ) const override;
};
```

- A derived **Piece** class
- It defines the **validMove** method

Pawn…: chess.[h/cpp]

```
class Pawn : public Piece
{
protected:
	friend PieceFactory< Pawn >;
	Pawn( Player owner , int id ) : Piece( owner , id ) {}
public:
	int validMove( Position start , Position end , const Board& board ) const override;
};
```

- A derived Piece class
- It defines the validMove method
- It has a protected constructor that invokes the base constructor
  - It explicitly grants the PieceFactory< Pawn > access to its constructor

ChessGame: chess.[h/cpp]

```
class ChessGame : public Board
{
public:
	ChessGame( void ) : Board( 8 , 8 )
	{
		addFactory( new PieceFactory< Pawn >  ( PAWN_ENUM ) );
		addFactory( new PieceFactory< Rook >( ROOK_ENUM ) );
		…
	}
	virtual void setupBoard( vid );
	virtual bool gameOver( void ) const override;
	virtual int makeMove( Position start , Position end ) override;
	virtual int run( void );
};
```

- A derived **Board** class for chess

ChessGame: chess.[h/cpp]

```
class ChessGame : public Board
{
public:
	ChessGame( void ) : Board( 8 , 8 )
	{
		addFactory( new PieceFactory< Pawn >( PAWN_ENUM ) );
		addFactory( new PieceFactory< Rook >( ROOK_ENUM ) );
		...
	}
	virtual void setupBoard( vid );
	virtual bool gameOver( void ) const override;
	virtual int makeMove( Position start , Position end ) override;
	virtual int run( void );
};
```

- A derived **Board** class for chess
  - Using an 8x8 board with registered factories for the chess **Pieces**

ChessGame: chess.[h/cpp]

```
class ChessGame : public Board
{
public:
	ChessGame( void ) : Board( 8 , 8 )
	{
		addFactory( new PieceFactory< Pawn >  ( PAWN_ENUM ) );
		addFactory( new PieceFactory< Rook >( ROOK_ENUM ) );
		…
	}
	virtual void setupBoard( vid );
	virtual bool gameOver( void ) const override;
	virtual int makeMove( Position start , Position end ) override;
	virtual int run( void );
};
```

- A derived **Board** class for chess
- Provides functionality for setting up the initial positions

ChessGame: chess.[h/cpp]

```
class ChessGame : public Board
{
public:
	ChessGame( void ) : Board( 8 , 8 )
	{
		addFactory( new PieceFactory< Pawn >  ( PAWN_ENUM ) );
		addFactory( new PieceFactory< Rook >( ROOK_ENUM ) );
		…
	}
	virtual void setupBoard( vid );
	virtual bool gameOver( void ) const override;
	virtual int makeMove( Position start , Position end ) override;
};
```

- A derived **Board** class for chess
- Provides functionality for setting up the initial positions
- You do the rest

Piazza → Resources section → Resources tab → Exercise 12-1