Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik I (D-ITET)

Group 7, 17:00-19:00, ETZ E9

slides by: Xenofon Floros

Final Session (19/12/2011)

Series 11 – Inheritance

```
// matchesOO.h

// enum for the player type
enum player { empty, player1, player2 };

// struct for a move
struct move { int sx,sy,dx,dy; };

class Board {
  private:
      player board[3][3];
      static const char symbol[3];
  public:
    Board();
    void print() const;
    int getVal(unsigned int x, unsigned int y) const;
    void setVal(unsigned int x, unsigned int y, player p);
    bool isValidMove(player p, move m);
    bool hasReachedOtherSide(player p);
};

const char Board::symbol[3] = {' ', 'X', 'O'};
```

Series 11 – Inheritance

```
// matchesOO.h
class AbstractPlayer {
  private:
    player p;
  protected:
    int getValidMoves(Board *board, move* moves);
  public:
    player getPlayer() const;
    AbstractPlayer(player p);
    virtual ~AbstractPlayer() {};
    virtual bool makeMove(Board *board) { return false; };
};
class RandomPlayer : public AbstractPlayer {
  public:
    RandomPlayer(player p) : AbstractPlayer(p) {};
    virtual ~RandomPlayer() {};
    virtual bool makeMove(Board *board);
};
class SmartPlayer : public AbstractPlayer {
  public:
   SmartPlayer(player p) : AbstractPlayer(p) {};
   virtual ~SmartPlayer() {};
   virtual bool makeMove(Board *board);
};
```

Series 11 – Inheritance

```
// matchesOO.h

class HumanPlayer : public AbstractPlayer {
  public:  HumanPlayer(player p) : AbstractPlayer(p) {};
  virtual ~HumanPlayer() {};
  virtual bool makeMove(Board *board);
};

// class definition of Game class

class Game {
  private:
    AbstractPlayer *p1, *p2;
    Board board;
  public:
    Game(AbstractPlayer* p1, AbstractPlayer* p2);
    AbstractPlayer* play();
};
```

Series 11 – Inheritance

```
// matchesOO.cpp

bool Board::isValidMove(player p, move m) {
  // variables:
  // - direction: upwards (-1) for player 1, downwards (1) for player2
  // - opponent: other player
  int dir = p == player1 ? -1 : 1;
  player opponent = p == player1 ? player2 : player1;
  // eliminate impossible moves
  // - wrong forward movement
  // - start position does not contain current player
  if(m.sy + dir != m.dy || board[m.sx][m.sy] != p) return false;
  // correct moves
  // - straight ahead
  // - strike left and right
  if(m.sx == m.dx && board[m.dx][m.dy] == empty) return true;
  if(m.sx - 1 == m.dx && board[m.dx][m.dy] == opponent) return true;
  if(m.sx + 1 == m.dx && board[m.dx][m.dy] == opponent) return true;

return false;
}
```

Series 11 – Inheritance

```
// mathcesOO.cpp

// getValidMoves: compute all valid moves. return count.
int AbstractPlayer::getValidMoves(Board *board, move* moves) {
  int count = 0;
  // loop over all combinations of fields (start = sx,sy ; destination = dx,dy)
  // (this is a very bad algorithm in terms of runtime complexity, but good enough for here:-)
 for(int sx=0; sx<3; sx++) {
    for(int sy=0; sy<3; sy++) {
      for(int dx=0; dx<3; dx++) {
        for(int dy=0; dy<3; dy++) {
          move m = {sx,sy,dx,dy};
          if(board->isValidMove(getPlayer(), m)) {
            moves[count].sx = sx;
            moves[count].sy = sy;
            moves[count].dx = dx;
            moves[count].dy = dy;
            count++;
          }
        }
      }
    }
  }
  return count;
}
```

Series 11 – Inheritance

```
// matchesOO.cpp

// smart makeMove: perform a smart, valid move.
bool SmartPlayer::makeMove(Board *board) {
  // reserve space for all possible moves
  move* moves = new move[81];
  int count = getValidMoves(board, moves);
  if(count > 0) {
    // choose winning move, if any
    int moveChosen = 0;
    for(int i=0; i<count; i++) {
      if(moves[i].dy == 0 || moves[i].dy == 2) {
        moveChosen = i;
        break;
      }
    }
    board->setVal(moves[moveChosen].sx, moves[moveChosen].sy, empty);
    board->setVal(moves[moveChosen].dx, moves[moveChosen].dy, getPlayer());
    return true;
  } else {
    return false;
  }
}
```

Series 11 – Inheritance

```
// matchesOO.cpp

// random makeMove: perform a random, valid move.
bool RandomPlayer::makeMove(Board *board) {
  // reserve space for all possible moves
  move* moves = new move[81];
  int count = getValidMoves(board, moves);
  if(count > 0) {
    // perform move
    int moveChosen = rand() % count;
    board->setVal(moves[moveChosen].sx, moves[moveChosen].sy, empty);
    board->setVal(moves[moveChosen].dx, moves[moveChosen].dy, getPlayer());
    return true;
  } else {
    return false;
  }
}
```

Series 11 – Inheritance

```
// matchesOO.cpp

// human makeMove: perform an entered move.
bool HumanPlayer::makeMove(Board *board) {
  move m;
 // get start and destination field
 do {
    cout << "Start column:        ";
    cin >> m.sx;
    cout << "Start row:           ";
    cin >> m.sy;
    cout << "Destination column:  ";
    cin >> m.dx;
    cout << "Destination row:     ";
    cin >> m.dy;
  } while(!board->isValidMove(getPlayer(),m));
  // perform move
  board->setVal(m.sx, m.sy, empty);
  board->setVal(m.dx, m.dy, getPlayer());
  return true;
}
```

Series 11 – Inheritance

```
// matchesOO.cpp
AbstractPlayer* Game::play() {
  AbstractPlayer* winner = NULL;
  board = Board();
  AbstractPlayer* curPlayer = p1;
  int i=10;
  while(i>0) {
    if(curPlayer->makeMove(&board)) {
      board.print();
    } else {
      winner = curPlayer == p1 ? p2 : p1;
      cout << "Player " << curPlayer->getPlayer() << " blocked! Player " << winner->getPlayer() << " has
     won." << endl;
      break;
    }
    if(board.hasReachedOtherSide(curPlayer->getPlayer())) {
      winner = curPlayer;
      cout << "Player " << curPlayer->getPlayer() << " has won." << endl;
      break;
    }
    curPlayer = curPlayer == p1 ? p2 : p1;
    i--;
  }
return winner;
}
```

Exam Issues

- **1 hour written examination**.
- Any documentation is allowed – no electronics.
- You have to be fast !!
- Read the questions carefully in order to answer exactly and only what you are asked to.
- Go through the exercises you solved during the semester and try to understand your errors.
- Try to solve yourselves the previous years exams you are given, in order to get an estimate of the complexity and time needed.

- Your code should match the **standard C++ syntax** .

(no pseudo-code will be accepted, unless stated otherwise in the exercise)

- There are different levels of mistakes in the code you provide:

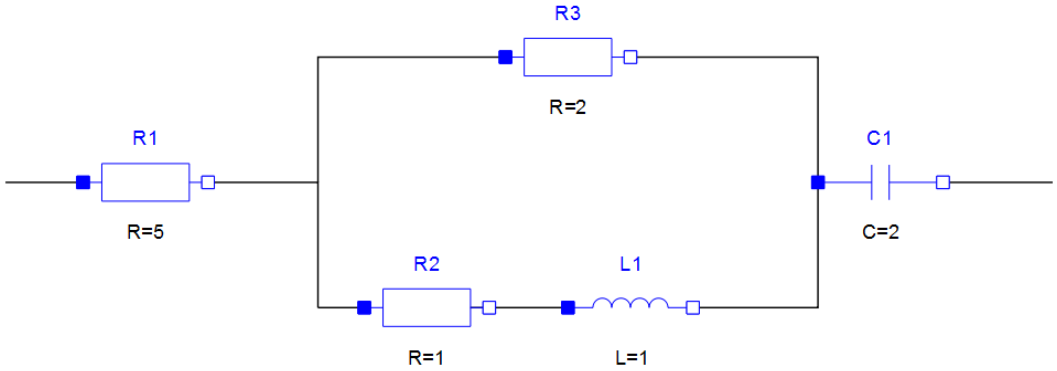
Forgotten semicolons (;) will not cost you as much as if you forget to dereference a pointer.

- It's better to completely answer a few questions, than partially cover more.

Christmas Exercise - 1

- In this exercise you are required to calculate the impedance of an RLC circuit, given as an XML file
- Examples:

```
<circuit>
    <series>
        <resistance>"5"</resistance>
        <parallel>
            <resistance>"2"</resistance>
            <series>
                <resistance>"1"</resistance>
                <coil>"1"</coil>
            </series>
        </parallel>
        <condenser>"2"</condenser>
    </series>
</circuit>
```



Christmas Exercise - 1

- The impedance can be complex, so we need to implement a Complex class with standard operations (+-*/) and possibly printing - could also be a struct but a class is much better.
- An easy way to perform the calculations is to use a stack – we push on a begin tag (e.g. <parallel>) and pop on an end tag (e.g.</parallel>) – you can use the stl stack for that or one of your older implementations

Christmas Exercise - 2

- In this exercise you will be implementing a simple soduko solver
- A soduko board is a 9x9 square of integers in the range 1-9
- At the beginning, some of the cells are empty, the objective is to fill all the cells with numbers according to the following rule:
  - Each line, column and quadrant (an aligned 3x3 block) must have exactly one of each of the numbers in the range 1-9

6	9	8	5	2	4	3	1	7
5	2	4	3	1	7	6	9	8
3	1	7	6	9	8	5	2	4
9	8	6	2	4	5	1	7	3
2	4	5	1	7	3	9	8	6
1	7	3	9	8	6	2	4	5
8	6	9	4	5	2	7	3	1
4	5	2	7	3	1	8	6	9
7	3	1	8	6	9	4	5	2

Christmas Exercise - 2

- You are given a template that provides random soduko boards, and checks whether a solution is valid

- A simple way to search is to start with all options [1..9] for each cell
- One by one you add the given numbers, eliminating from the row+column+region this number
- Whenever a cell is eliminated to one option, you set that cell to that number, using the previous algorithm
- If, at the end, there are unresolved numbers – no solution was found
- If you eliminate a cell to 0 options, then ?