An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 24

Revision

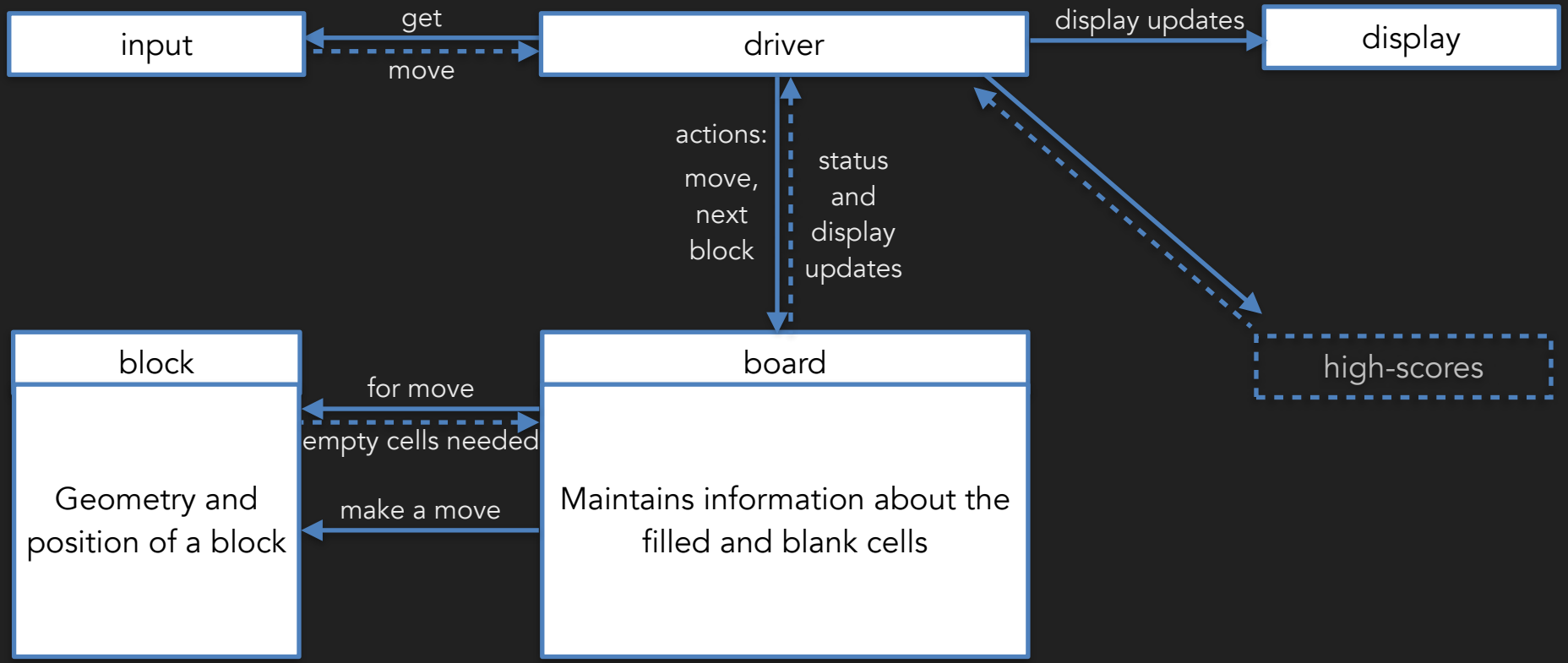
An Example: Designing a Large-ish Program

Based on material developed by Prof. Abhiram G. Ranade

Designing a Large Program

- Today a game of Tetris (see demo)
- Plan
  - A high-level design
  - Glimpses of various classes

Our High-Level Design



Some Low-Level Types

- For the different objects to communicate amongst themselves, need appropriate types

```
enum block_t {nil, I, O, L, J, T, S, Z }; // the 7 types of blocks, and empty
enum move_t {none, drp, dwn, rotCW, rotACW, lft, rgt}; // 6 types of moves + none

// the 4 orientations
class direction {
    char d=0; // direction is 0,1,2,3
public:
    direction& operator++() { (++d) %= 4; return *this;}
    direction& operator--() { (d+=3) %= 4; return *this;} // prevents going -ve
    bool operator==(const int& i) { return d==i;}
};
```

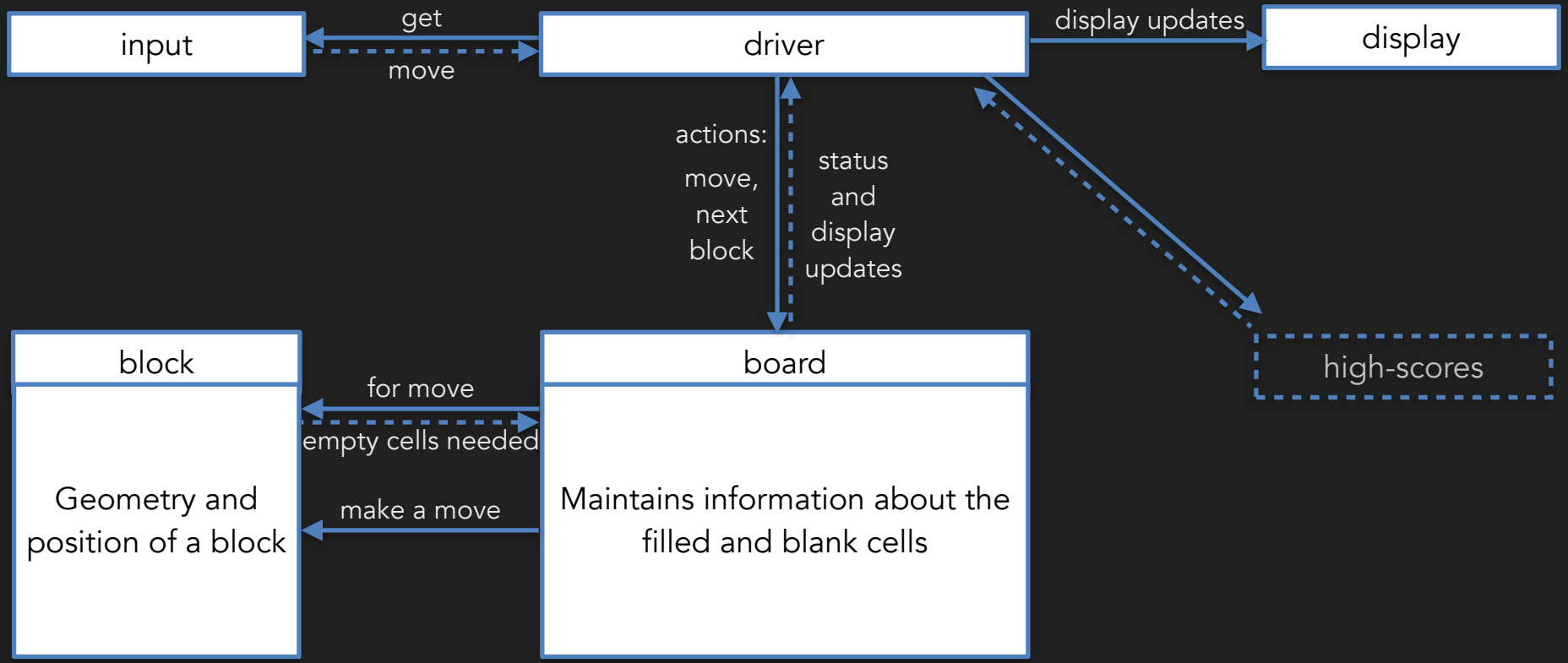
Some Low-Level Types

```
class point; // forward declaration
// a tetromino has 4 squares. stored tersely as a 2D array
struct tetromino {
    char pts[4][2];                 // the position of the 4 squares
    operator vector<point> () const; // to cast into a vector of 4 points
};
class point { // stores x, y coordinates of a cell in the board's grid
    char x, y;
public:
    point(char a=0, char b=0) : x(a), y(b) {}
    point(move_t mv); // calculate shift of origin due to a move
    point& operator+=(const point& p) { x += p.x; y += p.y; return *this; }
    tetromino operator+(const tetromino& t) const; // point+tetromino => tetromino
    bool operator==(const point& pt) const { return x==pt.x && y==pt.y; }
    int X() const { return x; }
    int Y() const { return y; }
};
```

Some Low-Level Types

```
// a class used to store a list of cells to be updated
// this will be used by the board class internally
class updateList {
    vector<point> Lpt;     // for each point in this list
    vector<block_t> Lblk;  // set it to be from this block type
public:
    void clear() { Lpt.clear(); Lblk.clear(); }
    size_t size() { return Lpt.size(); }
    int X(int i) { return Lpt[i].X(); }
    int Y(int i) { return Lpt[i].Y(); }
    block_t blk(int i) { return Lblk[i]; }
    // add updates corresponding to moving a block from start to finish
    void move(const tetromino& start, const tetromino& finish, block_t blk);
    // after extracting points, the list gets reset
    void extractPoints(vector<point>& up);
};
```

Our High-Level Design

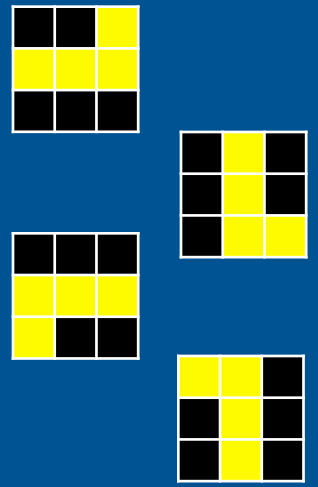


Block

```
class block {
    // static functions for computing geometry and room for manoeuvring
    static tetromino pattern(block_t,direction);
    static vector<point> relRoom(block_t, direction d, move_t);
    block_t blk;  // the type of the block
    point origin; // left-bottom corner of block's grid on the board (can be -ve)
    direction dir; // orientation of the block within its grid
    tetromino occupied; // cells occupied, relative to origin
public:
    block() = default; // needed to create an uninitialised block
    block(block_t b, point o)
          : blk(b), origin(o), occupied(origin + pattern(blk,dir)) {}
    block& operator= (const block& B);
    block_t blocktype() const { return blk; }
    tetromino where() const { return occupied; }
    vector<point> room(move_t mv); // list of squares needed to make a move
```

Shape of a Block

```
tetromino block::pattern(block_t b, direction d) {
    if(b==L) {
        if (d==0)
            return tetromino {{ {0,1}, {1,1}, {2,1}, {2,2} }};
        if (d==1)
            return tetromino {{ {1,0}, {1,1}, {1,2}, {2,0} }};
        if (d==2)
            return tetromino {{ {0,0}, {0,1}, {1,1}, {2,1} }};
        //if (d==3)
        return tetromino {{ {0,2}, {1,0}, {1,1}, {1,2} }};
    }
    ...
    if(b==O) {
        return tetromino {{ {0,0}, {1,0}, {0,1}, {1,1} }};
    }
    throw std::invalid_argument("Invalid block"); // for nil block type
};
```



Room for a Block

```
// for each block type, for each move, room needed to make the move mv
// relative to its origin
// This may include the room currently occupied as well
vector<point> block::relFullRoom(block_t b, direction d, move_t mv);

// remove the points occupied by the block from the room
vector<point> block::relRoom(block_t b, direction d, move_t mv) {
    auto room = relFullRoom(b,d,mv);
    auto self = pattern(b,d);
    for(int i=0; i<4; i++) { // there are 4 points in a tetromino
        point p(self.pts[i][0],self.pts[i][1]);
        auto it = std::find(room.begin(),room.end(),p);
        if(it!=room.end())
            room.erase(it);
    }
    return room;
}
```

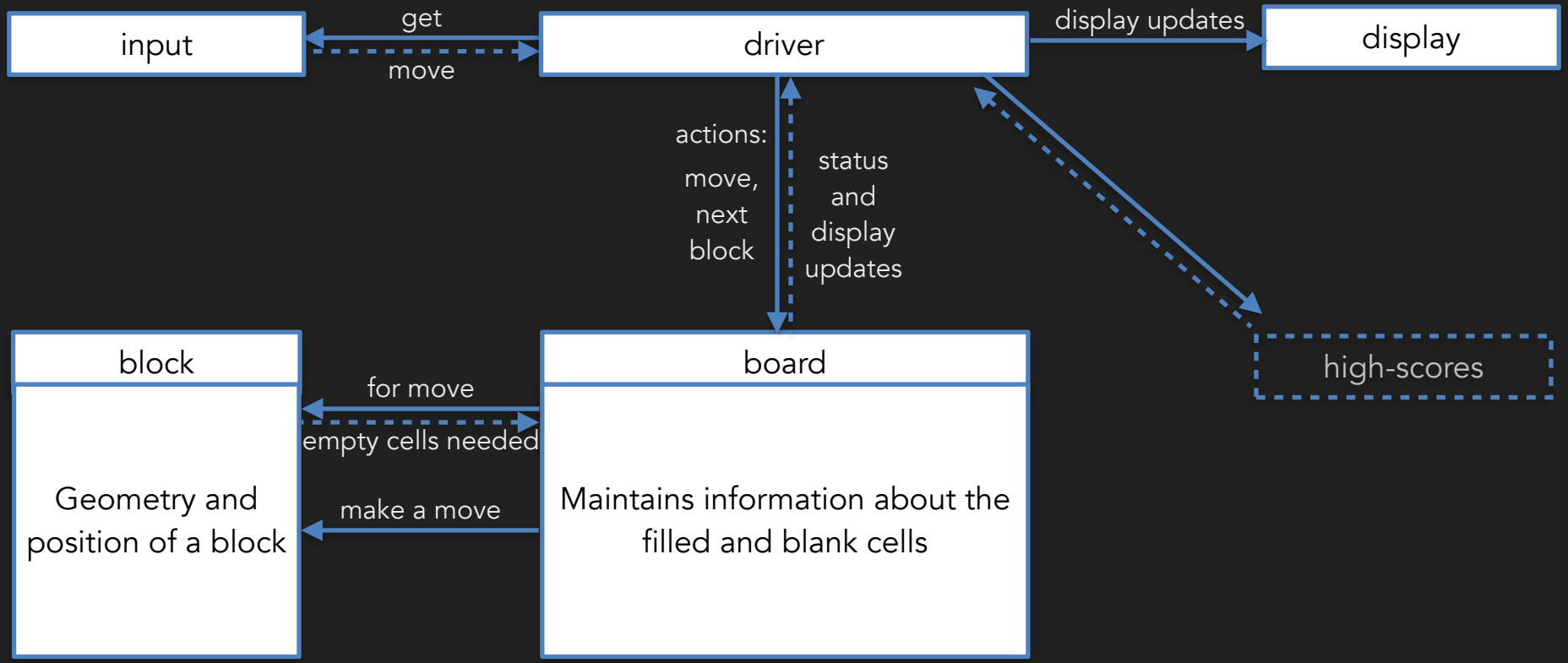
See code

```
vector<point> block::room(move_t mv) {
  auto v = relRoom(blk,dir,mv);
  for(auto& p: v) p += origin;
  return v;
}
```

Board

```
class board {
    int cols, rows;
    vector<vector<block_t>> blocks ; // blocks[y][x] block_t for y-th row, x-th col
    bool check(const vector<point>& v); // check if all points given are empty
    bool drop(block& B);                // drop (update B too). true if moved
    bool tryMove(block& B, move_t mv);  // move (for mv!=drp) if room available
    void applyUpdate(updateList& up);
public:
    board(int c,int r) : cols(c), rows(r), blocks(rows,vector<block_t>(cols,nil)) {}
    int ncols() const {return cols;}
    int nrows() const {return rows;}
    block_t at(int x, int y) const { return (blocks.at(y)).at(x);  }
    bool newBlock(block_t b, block& B); // return if success; B can start falling
    bool move(block& B, move_t mv, vector<point>& updates); // false if landed
    vector<int> collapse();    // collapse full rows. return their original indices
};
```

Our High-Level Design



Drawing the Board

- Using simplecpp graphics

```
class boardDrawer {
    const board* Brd; // will only use nrows(), ncols() and at() from board
    int nrows, ncols;
    double boxsz, height, width;
    vector<vector<Rectangle*>> boxes; // for each cell, the coloured box
    Rectangle* mkBox(int row, int col);
    Color getColor(int x, int y); // colour of board's block
public:
    boardDrawer (double height, int hiddenrows, const board* B);
    void refresh();  // redraw all the cells based on Brd
    void update(const vector<point>& L); // redraw only the cells listed in L
    void updateBlock(const block& piece); // redraw cells for a block
};
```

such a list is given by `board::move()`

Driving the Game

```
class game {
   // configuration parameters
    const int nPolls = 15;  // maxium number of inputs within one unit_time
    const double unit_time = 1,  blink_time = 0.5;
    const double speedup = 0.0005;
    board Tet;
    boardDrawer drawer;
    playerInput input;
public:
    game(int wd=10, int ht=22, int hid=2) : Tet(wd,ht), drawer(500, hid, &Tet) {}
    block_t nextBlock() {
        static dice<short> D(1,nBlocks);
        return static_cast<block_t>(int(D.roll()));
    }
    int play();
};
```

See code

Input

- simplecpp uses Xlib
  - KeySym is a data for storing a key

```
class playerInput {
    const KeySym DOWNKEY = XK_d, LEFTKEY = XK_Left, RIGHTKEY = XK_Right,
                 ROTCWKEY = XK_Up, ROTACWKEY = XK_Down, DROPKEY = XK_space;
    bool validKey(XKeyEvent& ev);
    move_t moveFromKey(XKeyEvent& ev);
public:
    void flushMoves(){ XEvent ev; while(checkEvent(ev)) {} }
    move_t nextMove();
};
```