A Boustrophedon for Chompers

Patrick Fleckenstein

Id: intro.nw,v 1.1 2000/03/07 20:15:41 pat Exp

Contents

1 Introduction **2**

2 Basic Strategy **3**

3 Questionable Design Decisions **3**

3.1 Dealing with Wrap-around . . . 3
3.2 The Array Reference in the Boustrophedon . . . 4

4 The Boustrophedon **5**

4.1 The Enumerated Type for Directions . . . 5
4.2 The Data Members . . . 5
4.2.1 The Version Identifier . . . 5
4.2.2 The Array . . . 5
4.2.3 The Turn Counter . . . 6
4.2.4 The Current Position . . . 6
4.2.5 The Direction of Motion . . . 6
4.2.6 The Turning Points . . . 6
4.3 The Methods . . . 7
4.3.1 The Boustrophedon Constructor . . . 7
4.3.2 The Boustrophedon Destructor . . . 7
4.3.3 Setting the Position . . . 7
4.3.4 Taking a Turn . . . 8
4.4 The Source Code . . . 13
4.4.1 The `boustrophedon.hh` . . . 13
4.4.2 The `boustrophedon.cc` . . . 13

5 The Array **14**

5.1 The Data Members . . . 14
5.1.1 The Version Identifier . . . 14
5.1.2 The Width and Height . . . 14
5.1.3 The Cells of the Array . . . 14
5.2 The Array Methods . . . 14
5.2.1 Reading in an Array . . . 14

5.2.2 The Array Constructor 15
5.2.3 The Array Destructor 16
5.2.4 The Width and Height Accessor Methods 16
5.2.5 The Cell Accessors 17
5.3 The Source Code 18
5.3.1 The `array.hh` 18
5.3.2 The `array.cc` 18

6 The Main Program 19
6.1 Processing the Initial Game Message 19
6.2 Procesing Each Turn 20
6.3 The Source Code 21
6.3.1 The `pc2.cc` 21

1 Introduction

Chompers is the problem for the Second Computer Science House Programming Competition[1]. In Chompers, all of the entrants compete simultaneously on the same playing field. The playing field is a simple array of letters. Entrants attempt to accumulate the most points. An entrant receives points for each letter it chomps. The higher the letter in the alphabet, the more points it is worth. Play continues until the array is empty or all player starve.

There is one other kink to this problem. The array wraps around at the edges. It does not, however, wrap around in the standard toroidal way. The array is mapped onto a projective sphere. When an entrant leaves the edge of the array, it returns to the array at the point diametrically opposed to where it left the board.

[1] `http://www.csh.rit.edu/~pat/pc`

2 Basic Strategy

This Chomper contestant is a simple fellow. It is boustrophedon[2]. It simply marches across the array in zig-zag fashion, chomping any time it gets to a square which has not yet been chomped. The only time it takes advantage of the projective sphere is in going from the last slow-scan edge back to the first slow-scan edge.

When it receives its initial position in the array, this entrant determines which direction to head first. Because of the projective-sphere wrap-around, we can eliminate walking in the same place twice if we choose a dimension of odd size as our slow-scan direction. The fast-scan direction can be either direction perpendicular to the slow-scan direction.

But, if both dimensions are even, then the best possible direction we can start in is the one that gets us to the edge the fastest. This will be our initial fast-scan direction. One of the two directions perpendicular to the fast-scan direction is chosen to be the slow-scan direction.

This entrant heads off in the fast-scan direction. Any time it comes to a cell in the array which has not been chomped, it chomps.

When this entrant reaches an edge, instead of continuing on in the fast-scan direction, it takes one step in the slow-scan direction and reverses its fast-scan direction. The only exception to this is when it crosses the edge in the slow-scan direction, it does not reverse its fast-scan direction.

This method of traversal ensures that every cell in the array is reached. The only cells that will ever be crossed twice before the array is empty are those cells from the starting point to the initial edge. That is why the fast-scan direction is initially chosen to be toward the closest edge. This ensures that this algorithm will tread twice over the smallest number of cells.

3 Questionable Design Decisions

3.1 Dealing with Wrap-around

I considered making a class which inherited from the `Array` class of section 5 that abstracted out the wrap-around features of the array. It initially seemed like it would be a good design decision because then hopefully the rest of the code could be used as-is if suddenly the problem changed to some wrap-around other than the projective-sphere wrap-around.

However, the simple zig-zag approach described above has a serious problem deciding where to go next from the slow-scan edge if there is no wrap-around and the same sort of problem at the non-wrapping edge of an array with Mobius wrap-around. It could easily miss a large number of cells in an array with spherical wrap-around. And, it could be greatly simplified to only zig with a toroidal wrap-around or Klein-bottle wrap-around array. So, as the boustro-

[2] the writing of alternate lines in opposite directions

phedon approach only serves me well in a projective-sphere array, I opted to just handle it within the `Boustrophedon` class itself.

3.2 The Array Reference in the Boustrophedon

I debated simply passing in the `Array` reference to the `Boustrophedon` instance when requesting each turn. I left it this way to imply that if this were a multi-threaded program, it could be thinking about the array (albeit a potentially out-of-date version of it) at any point during the turn sequence.

The `Boustrophedon` has no real reason to think out-of-turn. Its calculations are simple enough that they should not really cause it to starve. But, I like the possibility of threading this at a later point without having to make a copy of the array each time or receive other updates on the state of the array.

4 The Boustrophedon

The "meat" of the `Boustrophedon` class is in the `TakeTurn()` method in section 4.3.4. You can jump straight to it if you're only interested in the guts of the algorithm.

4.1 The Enumerated Type for Directions

To make things more explicit, we use an enumerated type within the `Boustrophedon` to name the directions.

5a ⟨*boustrophedon protected enumerations* 5a⟩≡ (13a)

```
typedef enum {
    NORTH = 0,
    SOUTH = 1,
    EAST  = 2,
    WEST  = 3
} Direction;
```

4.2 The Data Members

4.2.1 The Version Identifier

To keep things absolutely clear, it is marked with a version identifier.

5b ⟨*boustrophedon version identifier* 5b⟩≡

```
"$Id: boustrophedon.nw,v 1.1 2000/03/07 20:15:39 pat Exp $"
```

This version identifier is kept as a static data member of the class.

5c ⟨*boustrophedon protected data declarations* 5c⟩≡ (13a) 5e▷

```
static const char* _versionID;
```

5d ⟨*boustrophedon static variables* 5d⟩≡ (13b)

```
const char* Boustrophedon::_versionID = ⟨array version identifier 14a⟩;
```

4.2.2 The Array

Here, we will keep a reference to the `Array` instance for the game. In the main loop in section 6.2, we will be updating the `Array` instance to reflect the chomps of all of the players.

5e ⟨*boustrophedon protected data declarations* 5c⟩+≡ (13a) ◁5c 6a▷

```
const Array& _arr;
```

Uses `Array` 18a.

4.2.3 The Turn Counter

Here, we keep track of how many times the `TakeTurn()` method has been called. We need to at least know when the first turn happens because we are unable to decide which fast-scan direction to choose until we know where in the array we are. We do not find that out until the first turn when the main loop in section 6.2 calls our `TakeTurn()` method.

6a ⟨*boustrophedon protected data declarations* 5c⟩+≡ (13a) ◁5e 6b▷
```
unsigned int _turnCount;
```

4.2.4 The Current Position

Here, we keep track of our current position in the array.

6b ⟨*boustrophedon protected data declarations* 5c⟩+≡ (13a) ◁6a 6c▷
```
unsigned int _x;
unsigned int _y;
```

4.2.5 The Direction of Motion

Here, we keep track of the fast-scan and slow-scan directions. These are used in the `TakeTurn()` method for decision-making.

6c ⟨*boustrophedon protected data declarations* 5c⟩+≡ (13a) ◁6b 6d▷
```
Direction _fastScan;
Direction _slowScan;
```

4.2.6 The Turning Points

At various points in zig-zag scanning, we have to change direction. If we were in a 10 by 5 array and our fast-scan direction were east, then we would have to change fast-scan direction in column 1. And, the slow-scan edge we are going to bump into is after row 5. Once we change fast-scan directions to west, our next change will be in column 10.

Here, we keep track of the fast-scan edge coordinate we will encounter next and the fast-scan edge coordinate we would encounter in the other fast-scan direction along with the slow-scan edge coordinate.

6d ⟨*boustrophedon protected data declarations* 5c⟩+≡ (13a) ◁6c
```
unsigned int _fastScanEdge;
unsigned int _otherFastScanEdge;
unsigned int _slowScanEdge;
```

4.3 The Methods

4.3.1 The Boustrophedon Constructor

With the `Boustrophedon` constructor, we first save the reference to the `Array` instance for the game. We also set the turn counter to zero so that we will know the first time the `TakeTurn()` method gets called.

7a ⟨*boustrophedon public method declarations* 7a⟩≡ (13a) 7c▷

```
Boustrophedon( const Array& arr );
```

Uses `Array` 18a.

7b ⟨*boustrophedon methods* 7b⟩≡ (13b) 7d▷

```
Boustrophedon::Boustrophedon( const Array& arr )
        : _arr( arr ), _turnCount( 0 )
{
}
```

Uses `Array` 18a.

4.3.2 The Boustrophedon Destructor

We have nothing in the `Boustrophedon` class that takes up any

7c ⟨*boustrophedon public method declarations* 7a⟩+≡ (13a) ◁7a 7e▷

```
~Boustrophedon( void );
```

7d ⟨*boustrophedon methods* 7b⟩+≡ (13b) ◁7b 7f▷

```
Boustrophedon::~Boustrophedon( void )
{
}
```

4.3.3 Setting the Position

In the main loop in section 6.2, we will be calling this method to update the position of the `Boustrophedon`.

7e ⟨*boustrophedon public method declarations* 7a⟩+≡ (13a) ◁7c 8a▷

```
void SetPosition( unsigned int x, unsigned int y );
```

7f ⟨*boustrophedon methods* 7b⟩+≡ (13b) ◁7d 8b▷

```
void
Boustrophedon::SetPosition( unsigned int x, unsigned int y )
{
    _x = x;
    _y = y;
}
```

4.3.4 Taking a Turn

Here is where we do all of the real work in the `Boustrophedon` class. If this is the first turn, we decide on a strategy. Otherwise, we simply go along with our zig-zag strategy.

8a ⟨*boustrophedon public method declarations* 7a⟩+≡ (13a) ◁7e

```
char TakeTurn( void );
```

8b ⟨*boustrophedon methods* 7b⟩+≡ (13b) ◁7f

```
char
Boustrophedon::TakeTurn( void )
{
    ⟨boustrophedon turn lookup tables 9a⟩
    char move;

    ⟨boustrophedon make strategy 10b⟩
    ⟨boustrophedon choose move 8c⟩
    ++_turnCount;

    return move;
}
```

If the spot we are on has food in it, then eat.

8c ⟨*boustrophedon choose move* 8c⟩≡ (8b) 8d▷

```
if ( _arr.GetCell( _x, _y ) != ' ' ) {
    move = 'C';
}
```

Otherwise, check to see if we've hit the fast-scan edge. If we have, then deal with that.

8d ⟨*boustrophedon choose move* 8c⟩+≡ (8b) ◁8c 8e▷

```
else if ( ⟨boustrophedon fast scan coordinate 9b⟩ == _fastScanEdge ) {
    ⟨boustrophedon deal with fast scan edge 9e⟩
}
```

Otherwise, just move in the fast-scan direction.

8e ⟨*boustrophedon choose move* 8c⟩+≡ (8b) ◁8d

```
else {
    move = moveName[ _fastScan ];
}
```

In order to quickly determine the fast-scan coordinates, we will be using a bunch of masking operations on the coordinates based on the fast-scan direction. The `xMasks` and `yMasks` tables keep the masks for each direction. The directions are in the order given in the `Direction` enum in section 4.1.

9a ⟨*boustrophedon turn lookup tables* 9a⟩≡ (8b) 9d ▷
```
static unsigned int xMasks[] = {
    0, 0, ~0, ~0
};

static unsigned int yMasks[] = {
    ~0, ~0, 0, 0
};
```

Then, to calculate the current position in the fast-scan direction, we can mask the current position with the fast-scan masks from these tables and combine the results.

9b ⟨*boustrophedon fast scan coordinate* 9b⟩≡ (8d)
```
( ( _x & xMasks[ _fastScan ] ) | ( _y & yMasks[ _fastScan ] ) )
```

We can do the analogous thing for the slow-scan direction.

9c ⟨*boustrophedon slow scan coordinate* 9c⟩≡ (9e)
```
( ( _x & xMasks[ _slowScan ] ) | ( _y & yMasks[ _slowScan ] ) )
```

Once we've decided on a direction to move, we have to translate that back into a named move. We use the `moveName` lookup table here to do that translation.

9d ⟨*boustrophedon turn lookup tables* 9a⟩+≡ (8b) ◁ 9a 10a ▷
```
static char moveName[] = {
    'N', 'S', 'E', 'W'
};
```

If we hit a fast-scan edge, then we have to also check to see if we are at the slow-scan edge. If we aren't at the slow-scan edge, then we need to reverse the fast-scan direction. Either way, we need to move in the slow-scan direction.

9e ⟨*boustrophedon deal with fast scan edge* 9e⟩≡ (8d)
```
if ( ⟨boustrophedon slow scan coordinate 9c⟩ != _slowScanEdge ) {
    unsigned int tEdge = _fastScanEdge;
    _fastScanEdge = _otherFastScanEdge;
    _otherFastScanEdge = tEdge;
    _fastScan = oppositeDir[ _fastScan ];
}

move = moveName[ _slowScan ];
```

For convenience, we also keep a lookup table of what directions are opposite given directions.

10a ⟨*boustrophedon turn lookup tables* 9a⟩+≡ (8b) ◁9d

```
static Direction oppositeDir[] = {
    SOUTH, NORTH, WEST, EAST
};
```

To decide upon a strategy, we must determine our initial fast-scan direction and then use that to determine our slow-scan direction and our edge boundaries.

10b ⟨*boustrophedon make strategy* 10b⟩≡ (8b)

```
if ( _turnCount == 0 ) {
    ⟨boustrophedon find fast scan direction 10c⟩
    ⟨boustrophedon set slow scan direction and edges 12⟩
}
```

As we mentioned before in section 2, if either dimension of the array is odd, then we can choose the fast-scan direction to be perpendicular to it.

But, if both dimensions of the array are even, it behooves us to start fast-scanning in the closest direction.

10c ⟨*boustrophedon find fast scan direction* 10c⟩≡ (10b)

```
if ( ( _arr.GetHeight() & 1 ) == 1 ) {
    _fastScan = EAST;
} else if ( ( _arr.GetWidth() & 1 ) == 1 ) {
    _fastScan = SOUTH;
} else {
    ⟨boustrophedon find direction to closest edge 11⟩
}
```

To find the direction to the closest edge, we simply calculate the distance to each of the edges. After that, we assume that north is the closest edge. Then, we go through each other edge seeing if any are closer. Every time we come to a closer one, we reset the fast-scan direction to the closer one and reset the minimum distance.

11 ⟨*boustrophedon find direction to closest edge* 11⟩≡ (10c)

```
unsigned int dn = _y - 1;
unsigned int ds = _arr.GetHeight() - _y;
unsigned int de = _arr.GetWidth() - _x;
unsigned int dw = _x - 1;
unsigned int dmin;

_fastScan = NORTH;
dmin = dn;

if ( ds < dmin ) {
    _fastScan = SOUTH;
    dmin = ds;
}
if ( de < dmin ) {
    _fastScan = EAST;
    dmin = de;
}
if ( dw < dmin ) {
    _fastScan = WEST;
    dmin = dw;
}
```

Once we have the fast-scan direction, we simply choose the slow-scan direction to be $\frac{\pi}{2}$-radians clockwise[3] from the fast-scan direction. Then, we set the fast-scan edge to be the first edge we will bump in the fast-scan direction, we set the other fast-scan edge to be the first edge we would bump in the direction opposite the fast-scan direction, and we set the slow-scan edge to be the edge we will bump into in the slow-scan direction.

12 ⟨*boustrophedon set slow scan direction and edges* 12⟩≡ (10b)

```
switch ( _fastScan ) {
case NORTH:
    _fastScanEdge = 1;
    _otherFastScanEdge = _arr.GetHeight();
    _slowScan = EAST;
    _slowScanEdge = _arr.GetWidth();
    break;
case SOUTH:
    _fastScanEdge = _arr.GetHeight();
    _otherFastScanEdge = 1;
    _slowScan = WEST;
    _slowScanEdge = 1;
    break;
case EAST:
    _fastScanEdge = _arr.GetWidth();
    _otherFastScanEdge = 1;
    _slowScan = SOUTH;
    _slowScanEdge = _arr.GetHeight();
    break;
case WEST:
    _fastScanEdge = 1;
    _otherFastScanEdge = _arr.GetWidth();
    _slowScan = NORTH;
    _slowScanEdge = 1;
    break;
}
```

[3]When viewed from a positive altitude

4.4 The Source Code

4.4.1 The boustrophedon.hh

13a ⟨*boustrophedon.hh* 13a⟩≡

```
class Boustrophedon {
protected:
    ⟨boustrophedon protected enumerations 5a⟩
    ⟨boustrophedon protected data declarations 5c⟩
public:
    ⟨boustrophedon public method declarations 7a⟩
};
```

4.4.2 The boustrophedon.cc

13b ⟨*boustrophedon.cc* 13b⟩≡

```
#include <iostream>

#include "array.hh"
#include "boustrophedon.hh"

⟨boustrophedon static variables 5d⟩
⟨boustrophedon methods 7b⟩
```

5 The Array

5.1 The Data Members

The Array class simply manages the input array.

5.1.1 The Version Identifier

To keep things absolutely clear, it is marked with a version identifier.

14a ⟨*array version identifier* 14a⟩≡ (5d 14c)
```
"$Id: array.nw,v 1.1 2000/03/07 20:15:37 pat Exp $"
```

This version identifier is kept as a static data member of the class.

14b ⟨*array protected data declarations* 14b⟩≡ (18a) 14d ▷
```
static const char* _versionID;
```

14c ⟨*array static variables* 14c⟩≡ (18b)
```
const char* Array::_versionID = ⟨array version identifier 14a⟩;
```
Uses `Array` 18a.

5.1.2 The Width and Height

We keep track of the width and height of the the array as unsigned integers in the `Array` class.

14d ⟨*array protected data declarations* 14b⟩+≡ (18a) ◁ 14b 14e ▷
```
unsigned int _width;
unsigned int _height;
```

5.1.3 The Cells of the Array

We store the actual array of cells as an array of characters. Note: Even though the input array is two-dimensional, it is stored here one-dimensionally. The accessor methods for the cells, which are described in section 5.2.5, take care of doing the mapping from two-dimensional coordinates to a one-dimensional index.

14e ⟨*array protected data declarations* 14b⟩+≡ (18a) ◁ 14d
```
char* _cells;
```

5.2 The Array Methods

5.2.1 Reading in an Array

We need some way to read the array in from an input stream. We provide the canonical form for this method:

14f ⟨*array public method declarations* 14f⟩≡ (18a) 15g ▷
```
friend istream& operator >> ( istream& in, Array& arr );
```
Uses `Array` 18a.

15a ⟨*array methods* 15a⟩≡ (18b) 16a▷
```
istream&
operator >> ( istream& in, Array& arr )
{
    ⟨array read in width and height 15b⟩
    ⟨array allocate the cells of the array 15e⟩
    ⟨array read in cells 15f⟩
    return in;
}
```
Uses `Array` 18a.

The array format is quite simple. There is a small header consisting of two integers. These are the width and the height of the array respectively.

15b ⟨*array read in width and height* 15b⟩≡ (15a) 15c▷
```
in >> arr._width >> arr._height;
```

We also check for problem cases.

15c ⟨*array read in width and height* 15b⟩+≡ (15a) ◁15b
```
if ( in.bad() || arr._width == 0 || arr._height == 0 ) {
    ⟨array set istream error flags 15d⟩
    return in;
}
```

To set error flags, we can really only tweak the flags in the input stream.

15d ⟨*array set istream error flags* 15d⟩≡ (15c)
```
in.setf( ios::badbit );
```

Once we know the size of the array, we allocate it.

15e ⟨*array allocate the cells of the array* 15e⟩≡ (15a)
```
arr._cells = new char[ arr._width * arr._height ];
```

Following the header are capital letters, one for each cell in the array. We ignore any whitespace.

15f ⟨*array read in cells* 15f⟩≡ (15a)
```
for ( unsigned int ii=0; ii < arr._width * arr._height; ++ii ) {
    in >> arr._cells[ ii ];
}
```

5.2.2 The Array Constructor

In the array constructor, we simply set things to an empty state. We zero out the width and height and null out the cells array.

15g ⟨*array public method declarations* 14f⟩+≡ (18a) ◁14f 16b▷
```
Array( void );
```
Uses `Array` 18a.

16a ⟨*array methods* 15a⟩+≡ (18b) ◁15a 16c▷
```
Array::Array( void )
{
    _width = 0;
    _height = 0;
    _cells = 0;
}
```
Uses `Array` 18a.

5.2.3 The Array Destructor

In the array destructor, we delete the array of cells.

16b ⟨*array public method declarations* 14f⟩+≡ (18a) ◁15g 16d▷
```
~Array( void );
```
Uses `Array` 18a.

16c ⟨*array methods* 15a⟩+≡ (18b) ◁16a 17b▷
```
Array::~Array( void )
{
    delete[] _cells;
}
```
Uses `Array` 18a.

5.2.4 The Width and Height Accessor Methods

We provide simple accessor methods for the width and the height of the array of cells.

In the accessor for the width, we simply return the width.

16d ⟨*array public method declarations* 14f⟩+≡ (18a) ◁16b 16e▷
```
inline unsigned int GetWidth( void ) const
{
    return _width;
}
```

Unsurprisingly, in the accessor for the height, we simply return the height.

16e ⟨*array public method declarations* 14f⟩+≡ (18a) ◁16d 17a▷
```
inline unsigned int GetHeight( void ) const
{
    return _height;
};
```

5.2.5 The Cell Accessors

With these accessor methods, we allow retrieval and modification of particular cells. We use x and y coordinates to specify particular cells. We assume that the x coordinate is on the range $[1,$ `_width`$]$ and the y coordinate is on the range $[1,$ `_height`$]$.

Because the cells are stored in a one-dimensional array, we must mingle the two-dimensional coordinates into a single index. We stored the cells left-to-right and top-to-bottom. So, the first `_width` items in the array are the cells of the first row of the array. The next row of cells begins after the first row at the `_width`-th cell. In general, the $y\$ - throwstartsthe$(y-1)*$`_width` cell in the `_cells` array.

For the `GetCell()` method, we simply calculate the one-dimensional index for the given two-dimensional coordinates and return the item in that cell.

17a ⟨*array public method declarations* 14f⟩+≡ (18a) ◁16e 17c▷

```
char GetCell( unsigned int x, unsigned int y ) const;
```

17b ⟨*array methods* 15a⟩+≡ (18b) ◁16c 17d▷

```
char
Array::GetCell( unsigned int x, unsigned int y ) const
{
    unsigned int index;
    index = ( y - 1 ) * _width + ( x - 1 );
    return _cells[ index ];
}
```

Uses `Array` 18a.

For the `SetCell()` method, we simply calculate the one-dimensional index for the given two-dimensional coordinates and replace the item in that cell with the new item.

17c ⟨*array public method declarations* 14f⟩+≡ (18a) ◁17a

```
void SetCell( unsigned int x, unsigned int y, char newItem );
```

17d ⟨*array methods* 15a⟩+≡ (18b) ◁17b

```
void
Array::SetCell( unsigned int x, unsigned int y, char newItem )
{
    unsigned int index;
    index = ( y - 1 ) * _width + ( x - 1 );
    _cells[ index ] = newItem;
}
```

Uses `Array` 18a.

5.3 The Source Code

5.3.1 The array.hh

18a ⟨*array.hh* 18a⟩≡

```
class Array {
protected:
      ⟨array protected data declarations 14b⟩
public:
      ⟨array public method declarations 14f⟩
};
```

Defines:
`Array`, used in chunks 5e, 7, 14–17, 19b, and 22.

5.3.2 The array.cc

18b ⟨*array.cc* 18b⟩≡

```
#include <iostream>

#include "array.hh"

⟨array static variables 14c⟩
⟨array methods 15a⟩
```

6 The Main Program

The main program is responsible for reading in the initial messages from the game server, reading the turn information from the game server, keeping the array up-to-date, asking the `Boustrophedon` instance for a move, and sending this move off to the game server. Once the `Boustrophedon` player is deceased, the main program exits.

6.1 Processing the Initial Game Message

The first line of the initial message from the game server contains the player number for the player represented by this process and the count of the total number of players in the game.

During initialization, we just cache these numbers away. We will use them again during the processing of each turn to know which turn information we should use to update the `Boustrophedon` instance's position in the array.

19a ⟨*main initialization* 19a⟩≡ (21) 19b ▷

```
unsigned int indexOfThisPlayer;
unsigned int playerCount;

cin >> indexOfThisPlayer >> playerCount;
```

Following that first line in the initial message is the starting array for the game. Here, we simply read that into an `Array` instance.

19b ⟨*main initialization* 19a⟩+≡ (21) ◁ 19a 19c ▷

```
Array arr;
cin >> arr;
```

Uses `Array` 18a.

Now, we create a `Boustrophedon` instance. We give the `Boustrophedon` constructor a reference to our `Array` instance so that it will always be able to see the current state of the array. We will be updating the array any time a player chomps.

19c ⟨*main initialization* 19a⟩+≡ (21) ◁ 19b

```
Boustrophedon bst( arr );
```

6.2 Procesing Each Turn

While the `Boustrophedon` player is still alive, we keep looping. On each pass through the loop, we read the moves all of the players made last turn. Be sure to update the `Array` and the `Boustrophedon`. Then, get the `Boustrophedon` move and send it out.

20a ⟨*main turn loop* 20a⟩≡ (21)

```
bool alive = true;

while ( alive ) {
    for ( unsigned int ii=0; ii < playerCount; ++ii ) {
        ⟨main loop read player move 20b⟩
        ⟨main loop update array 20c⟩
        ⟨main loop update boustrophedon 20d⟩
    }
    ⟨main loop emit boustrophedon turn 20e⟩
}
```

The information about a player's move is simply the current coordinates of the player and the move they last made.

20b ⟨*main loop read player move* 20b⟩≡ (20a)

```
unsigned int x;
unsigned int y;
char move;

cin >> x >> y >> move;
```

If the move was a chomp, then we have to clear out the spot in the array.

20c ⟨*main loop update array* 20c⟩≡ (20a)

```
if ( move == 'C' ) {
    arr.SetCell( x, y, ' ' );
}
```

And, if this was the `Boustrophedon`, then we must update its position. And, we also check to make sure our hero is still alive.

20d ⟨*main loop update boustrophedon* 20d⟩≡ (20a)

```
if ( ii+1 == indexOfThisPlayer ) {
    bst.SetPosition( x, y );
    alive = ( move != 'D' );
}
```

To emit the `Boustrophedon`'s turn, we simply send the return value from its `TakeTurn()` method and send it along the output.

20e ⟨*main loop emit boustrophedon turn* 20e⟩≡ (20a)

```
char move = bst.TakeTurn();
cout << move << flush;
```

6.3 The Source Code

6.3.1 The `pc2.cc`

21 ⟨*pc2.cc* 21⟩≡

```
#include <iostream>

#include "array.hh"
#include "boustrophedon.hh"

int
main( void )
{
    ⟨main initialization 19a⟩

    ⟨main turn loop 20a⟩

    return 0;
}
```

Revision History

22 ⟨*nsp revision history* 22⟩≡

```
$Log: appendices.nw,v $
Revision 1.1  2000/03/07 20:15:34  pat
Added to CVS control

Revision 1.3  2000/03/01 22:51:01  pat
Working version.  The text is pretty complete.
This may be the final draft.

Revision 1.2  2000/03/01 04:11:42  pat
Fleshed out a great deal of stuff.
Added accessors to the Array class.
Made a constructor for the Boustrophedon.
Added the section on questionable design decisions.
Added some detail on the initialization in the main program.

Revision 1.1  2000/02/24 16:10:55  pat
Initial revision
```

Uses `Array` 18a.

Index

`Array`: 5e, 7a, 7b, 14c, 14f, 15a, 15g, 16a, 16b, 16c, 17b, 17d, 18a, 19b, 22