CMPT 212 (2008-1) Assignment 3*
Due online Tuesday, March 25, 2008, 11:00pm

Ján Maňuch
jmanuch@sfu.ca

1 Programming Assignment

Write a C++ library implementing a field of agents. Your library will be split into two files:

1. the header file `field.h` containing the declarations of classes/types `Agent` (virtual base class), `Point`, `Region` and `Field`, where the first 3 classes/types are predefined, while the last one needs to be implemented by you;

2. the source code file `field.cpp` containing the implementation of the class `Field`, that is definitions of all its class methods.

Note that none of your files should contain the `main()` function. The `main()` function will be provided in a *testing file* which will include (using `#include "field.h"` directive) your header file and will defined some concrete agents (derived from `Agent`) and use them together with the public class methods of the class `Field`. The executable file will be generated by linking together object file of `field.cpp` and the object file of the testing file.

Your program must compile under Visual Studio C++ 2005. You will be penalized if some extra work is needed to compile your program on Visual Studio C++ 2005.

All your code must be located in two files called `field.h` and `field.cpp`, which will be the only files submitted for the assignment. You can create your own testing file for testing your library, but do **not** send your testing file in your submission!

*** **Important** ***

You are not allowed to use any dynamic or static arrays, neither you are allowed to use `new` operator (or any of its C-style variants) to dynamically allocate memory! You have to use containers of STL to store more than one elements in a variable. You can use any STL library you want.

So, once again, you **cannot** declare arrays or allocate memory dynamically as follows:

```
int ar[20]; // static arrays not allowed in this assignment!
int* par=new ar[20]; // dynamic arrays not allowed!
Node *node=new Node; // dynamic memory allocation not allowed!
```

Instead, use a container, for instance `vector` to hold the values you need. For example,

```
vector<int> ar(20); // allowed
```

Information about evaluating and grading the assignment can be found at the course website.

If you have any questions about the assignment, do not hesitate to send me an email. I might revise the text of the assignment based on your questions, if I find out that some parts are difficult to understand or insufficiently specified. If you have troubles with implementing the assignment, you are welcome to see me or TA during office hours.

* *Revision* : 1.0 (Thursday 6th March, 2008,00:23); Details in this document may change before the date of the submission of the assignment. Please make sure you have the latest version.

2 Problem Specification

A field is a two-dimensional grid. Each cell of the grid has its position represented by `x` and `y` coordinates (stored in object of class `Point`). The lower left corner of the filed has coordinates `(0,0)`, the upper right corner `(sizex-1,sizey-1)`, where `sizex` and `sizey` are the parameters passed to the constructor of the field. Each cell of the field can contain several agents (any number from 0 to `LONG_MAX`) sorted by their priorities. More precisely, it can hold several pointers to agents (created by the user of the Field class). Each agent has a state (of type `char`) which can be obtained by calling method `state()` on the agent. Each cell has a state as well and this state is computed from the states of the agents in the cell as follows: The state of a cell is the state of the agent with the highest priority (or if there is no agent in the cell, the state is `' '`). Agents can see the states of 8 surrounding cells and also the list of states of all agents located in the same cell (sorted by priorities of the agents starting with the state of agent with highest priority).

2.1 Agents

Agent is an entity which at every time has some state, priority, can make clone of itself and mainly, can act to its environment. All agents will be derived from the *virtual base class* `Agent`:

```
class Agent {
 public:
  enum Action {STAY, DIE, MOVE, CLONE};
  // this defines a new type which can have only above possible values
  // (see p. 141 in the textbook for details)
  // to access the type or values you have to use scope resolution operator
  // (for instance, Agent::Action a=Agent::STAY)
  enum Dir {E, NE, N, NW, W, SW, S, SE};
  Agent() {}
  virtual ~Agent() {}
  virtual Agent *clone() const =0;
  virtual long priority() const =0;
  virtual char state() const =0;
  typedef pair<Action,vector<Dir> > Answer; // shortcut
  virtual Answer act(const vector<char> &surrounding,
                     const vector<char>& myposition,long level) =0;
};
```

Note the only time when the internal state of an agent can change is during execution of `act()` method, as all other methods are `const`. In this method, the agent gets information about its environment and based on this information changes its internal state and returns the description of actions which should be performed for this agent. The class `Field` is responsible for providing correct environment information for each agent it contains and performing the action requested by each agent (as described in details in a moment).

First, let's look at the environment information passed to the agent. As the first argument `surrounding` the agent expects a `vector` with 8 `char` values of states of all 8 neighboring cells of the agent in the field, see Figure 1. Recall that the state of the cell is the state of the agent located in the cell with the highest priority. If the cell is empty, the state is `' '` (a space). If the cell is outside of the field (i.e., if the agent is in the borderline cell), then the state is `'*'`.

As the second argument `myposition` the agent expects a `vector` of `char` values representing the states of all agents located in the same cell (including it's own state). The states should be sorted by priorities of these agents (obtained by calling `priority()` method for each agent) starting with the state of the agent with the highest priority. If there are agents with the same priority their relative order can be arbitrary. Finally, the third argument `level` should be the position of the current agent in the `myposition` list. Hence, it should be always true that `myposition[level]` has the same value as the method `state()` would return.

Example. Assume that the field contains agents $a1, \ldots, a10$ located as described in Figure 2. For simplicity assume that the priority returned by agent a`i` is `i`. Let's consider agent $a5$. What arguments should be passed to its `act()` method?

As `surround` it should get a vector of 8 `char`s containing values:

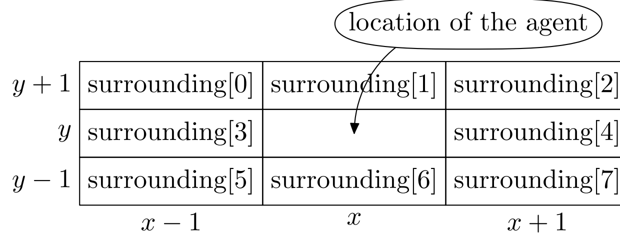

Figure 1: Description of what the `surrounding` argument of `Agent::act()` method should contain for the agent located in the cell with coordinates (x, y).

3		$a10$('w'), $a1$('j')		$a7$('+')
2	$a6$('o')	$a9$('k'), $a5$('v'), $a3$('p')	$a2$('q')	
1			$a8$('4')	
	0	1	2	3

Figure 2: Location of agents $a1, \ldots, a10$ in the field. The values in parenthesis are the current states of the agents.

`' ','w',' ','o','q',' ',' ','4'`.

For instance, the first value is the state of cell **`(0,3)`** which contains no agent, hence the state is a space **`' '`**, and the second value is the state of cell **`(1,3)`** which contains 2 agents $a10$ and $a1$. Since $a10$ has higher priority, the state of the cell is the state of $a10$, i.e., **`'w'`**.

As **`myposition`** it should get a vector of 3 **`char`**s containing values **`'k','v','p'`**, the states of all agents in the same cell (sorted by priorities of the agents, i.e., we start with $a9$ as it has highest priority 9 and finish with $a3$ as it has lowest priority 3). As **`level`**, it should get **`1`**, since $a5$ is in the second position (with index **`1`**) in the list of the agents in the cell **`(1,2)`** ordered by priorities.

On the other hand, agent $a6$ lies on the borderline, i.e., there are no cells to the left. Therefore, it will get **`surround`**:

`'*',' ','w','*','k','*',' ',' '`.

Let's look now at the return value of the **`act()`** method. It's a pair, where the **`first`** value is of type **`Action`** and the **`second`** is a sequence of **`Dir`** values. The meaning of the second returned value depends on the first returned value:

- **`STAY`**: agent should stay in the current cell, you can disregard the second returned value.
- **`DIE`**: agent should be removed completely from the field, you can disregard the second returned value. Remember to call **`delete`** on the pointer of the agent which is removed from the field to avoid memory leaks!
- **`MOVE`**: agent should move to one of the neighboring cells. The first element of the second returned value specifies the new cell where it should be moved. Note that if the agent would move outside of the field, it should die too (the agents in the test files shouldn't do that, unless you don't mark the cells which are outside of the field with **`'*'`** state in the **`surround`** argument).
- **`CLONE`**: agent should stay in the current cell, but new agents should be created using **`clone()`** method on this agent. The **`clone()`** method should be called as many times as is the number of elements of the second returned value (of type **`vector<Dir>`**). The new agents are placed in the neighboring locations specified by the second returned value.

  Note: The **`clone()`** method is supposed to be used **only** when creating new agents requested by **`CLONE`** returned value. Do not use **`new`** operator to create agents, just call the **`clone()`** method on the agent requesting to be cloned.

The variable of type `Dir` specifies a relative position of a neighboring cell to the current cell with a natural meaning. For instance, if the coordinates of the current cell (containing the agent) is (x, y), then the coordinates specified by value `SW` is $(x-1, y-1)$.

Example. Consider agent $a5$ from the previous example and assume it returns pair `Agent::CLONE` and a vector containing two values `Agent::W` and `Agent::NE`. Consequently, the `Field` class is suppose to leave $a5$ in it's place and create two new agents using `clone()` method on $a5$ and place them to cells `(0,2)` and `(2,3)` when processing agent $a5$.

2.2 Help classes/types

The file `field.h` contains the following definitions used in the interface of `Field` class:

```
class Point {
 public:
  int x,y;
  Point(int ix=0,int iy=0) : x(ix), y(iy) {}
};

typedef pair<Point,Point> Region;
```

Note the `Point` class contains its data members `x` and `y` in the `public` section, hence they can be easily accessed without calling any method. (In this case data hiding is not important: it's a very simple class and any values of `x` and `y` represent a meaningful state of `Point` object.)

`Region` is a pair specifying a rectangular region in the field by providing two `Point`s: the first (accessed via `first`) specifies the coordinate of the lower left cell of the region and the second (accessed via `second`) specifies of the coordinate of the upper right cell of the region incremented by `1` in each coordinate. For instance, to declare a variable of type `Region` containing all 12 cells depicted in Figure 2, you can write:

```
Region r(Point(0,1),Point(4,4));
```

2.3 Interface of Field Class

The class `Field` should implement the following *interface* described in the `public` section of the class.

```
class Field {
 private:
  // implementation details:
  // ...
 public:
  // public interface:
  Field(int sizex,int sizey);
    // construct field with no agents of size sizex x sizey
  ~Field(); // call delete on all pointers to agents stored in the field

  // ADDING AGENTS:
  void plant_agent(const Point &p, Agent *a);
  // put agent a to the cell p

  // EXAMINING AGENTS:
  long n_of_agents() const;
  // return the total number of agents currently on the field
  long n_of_agents(const Point &p) const;
  // return the number of agents in the cell p
  const Agent *get_agent(const Point &p, long i) const;
```

```
  // return a pointer to the i-th agent in the cell p
  // if i<0 or i>=n_of_agents(p) return NULL pointer

  // REMOVING AGENTS:
  void remove_agent(const Point &p, long i);
  // if i>=0 or i<n_of_agents(p) remove the agent (otherwise do nothing)
  // remember to call delete on the pointer to agent before removing it
  void remove_agent(const Point &p);
  // remove all agents from the cell p
  // remember to call delete on the pointer to each agent before removing it
  void remove_agent(const Region &r);
  // remove all agents in the region r
  // remember to call delete on the pointer to each agent before removing it
  // remember Region(Point(0,0),Point(2,2)) contains only cells
  // (0,0), (0,1), (1,0) and (1,1)!


  // EXAMINING FIELD:
  char get_state(const Point &p) const;
  // returns the state of agent with highest priority in cell p
  // or ' ' if there is no agent

  // !!! LIFE CYCLE (the main method) !!!:
  void step(long n=1);
  // performs n steps of life cycle of the field (described later in details)

  friend ostream & operator<<(ostream &os, const Field &s);
  // print the states of each cell in the region returned by range()
  // (see below)
};
```

2.4 Life cycle of the field

The main part of the class `Field` is to perform life cycle that is to call method `act()` on every agent on the field. Of course, before calling the `act()` method, it has to collect environment information for that particular agent. After calling method, it should perform the tasks requested by the agent (leave it as it is, remove it, move it to new cell or clone it), see Section 2.1.

The **important thing** is that the order in which you process the agents should not matter. In each step of the life cycle, the information about environment each agent gets should be the information before you start to make any changes to agent locations in that step. For instance, assume that you have a simple agent with state `'x'` at all times, acting as follows, if it sees `'x'` in any of the neighboring cells, it moves there, otherwise stays in its place. Now, assume your field contains only two such agents in neighboring positions. Then in the *correct* implementation, the program will swap the locations of the two agents in each step of the life cycle of the field. In the *incorrect* implementation, one agent moves the location of another and they both stay there in subsequent steps. Your implementation must have the correct behavior if you want to get any points from the assignment.

To achieve the correct behavior, use the following strategy:

- Collect environment information for each agent first in some temporary data structure, and only then start calling `act()` methods using recorded information and perform requested changes.

Remarks.

- Also make sure that you do not act on the agents which were just placed to the field using `clone()` method. They should start working only in the next step.

- Be careful when obtaining states of surrounding cells of an agent lying on the boundary of the field. This state should be `'*'`. Check that you are not accessing an element outside of declared space of any container you use to avoid incorrect behavior of your program and possibly runtime errors.

- In one step of the life cycle of the field the `act()` method should be called exactly once for each agent on the field! The `act()` method is changing internal state of the agents. For example, agent can count in which step of the life cycle it is depending on the number of calls to `act()` method. If for some reason, you would call `act()` several times in one step of the life cycle for one particular agent, the agent could be "confused" and the test program might behave differently than required (which will result in failing the test).

2.5 Printing function

The printing function (`operator<<()`) should print the states of all cells in the field. For each cell, it prints the state of the cell, i.e., the state of the agent with the highest priority, i.e., just one `char`. Each printed line should contain `sizex` characters (followed by a new line character) and the number of printed lines should be `sizey`. And remember when printing, you have to print lines for rows of the field with higher `y` coordinate first.

Example. For the field in Figure 2, assuming that the size of the field is 4×4, the print function should generate the following output (`\n` stands for a new line character as usual):

```
␣w␣+\n
okq␣\n
␣␣4␣\n
␣␣␣␣\n
```

2.6 Implementation

- The `private` section of the class `Field` should contain the implementation details:
  - *data members* representing the content of the set in memory — this is completely up to you, you can represent the set as an array containing the actual elements (in sorted or unsorted manner), or a linked list of all elements (in sorted or unsorted manner), etc.;
  - *auxiliary functions* which are not part of the interface but you need them to handle the data [you might not need any of such functions, it depends on your implementation].

- The `public` section of the class should contain only the interface described above. *You are not allowed to add or remove any functions to/from the interface, neither to modify the prototypes of these functions!* The `public` section should not contain any data members.

- The implementation of all member functions (auxiliary ones and the interface functions) should be contained in `field.cpp` file, not in `field.h` file. Do not add function bodies to the class declaration in `field.h` file.

- No other function than friend function `operator<<()` is supposed to print anything. Again, any other output from the program than required will make the evaluation process difficult and you will be penalized for that. In the function `operator<<()` remember to print to the stream and to `cout` and remember to return the stream at the end.

You don't have to type the interface again, here is the start-up header file: field.h and source code file: field.cpp (containing the definition of printing function for `Point` class).

3 Testing

A typical test file will derive a concrete agent class from the virtual base class `Agent`, plant some agents of this derived class to the field and call the `step()` method on the field to perform a certain number of steps. After that it will print the state of the field and test if it matches the expected state. The following are three simple agent classes which will test the basic abilities of your `Field` class.

3.1 Test application 1: The game of life

We have simple agents which do not move and clone and have only two states: `'.'` (dead state) and `'x'` (living state).

```
class Life : public Agent {
  char s; // state
 public:
  Life(char ini_state='.') : s(ini_state) {}
  Agent* clone() const { return new Life; } // not really needed
  long priority() const {return 0; }
  char state() const { return s; }
  Answer act(const vector<char> &surrounding,
             const vector<char>& myposition,long level)
  {
    // count the number of living neighbors
    int n=count(surrounding.begin(),surrounding.end(),'x');
    if (s=='.') {// dead
      if (n==3)
        s='x'; // go alive
    } else // alive
      if (n<2 || n>3)
        s='.'; // go dead
    return Answer(STAY,vector<Dir>());
  }
};
```

The complete test file can be downloaded here: a3_test1.cpp. The correct output of this program can be downloaded here: a3_test1-output.txt.

3.2 Test application 2: Running agents

The agents do not change states but simply move in a direction specified in the constructor in each step. This will test whether your field correctly resizes. The test file: a3_test2.cpp and the correct output: a3_test2-output.txt

3.3 Test application 3: Expansion

The agents clone in four directions in each step and also increment their state and priority. This will test whether your field correctly handles cloning. The test file: a3_test3.cpp and the correct output: a3_test3-output.txt.

The above three test files together with another three test files will be used to evaluate the assignment. All test files check for memory leaks (that all agents are freed from memory at the end).

4 Programming Contest

Assignment 3 will have an associated programming contest. Only assignments that are considered to have adequate implementations, as outlined in Section 2 will be entered into the contest.

The winners will be entries that result in the *shortest source code*. The size of your C++ code (both `field.h` and `field.cpp`) will not include any comments, spaces or `#include` directives. Hence, you should comment and format your source code as usual (otherwise you will get penalized in the clarity part of the assignment), and you can include as many STL libraries as you want. On the other hand, the length of used variables will affect the length of the source code. Therefore, you are allowed to use short names for variables (instead of descriptive long names) but make sure that your comments are sufficient to explain your code.

Up to three entries will be selected. In case of ties that result in more than three top entries of equal size, none of the tied entries will be considered.

The winners will be publicly acknowledged on the course website and during lecture. The source code of winning assignments will be uploaded to and linked from the course web site. Students who are not comfortable with these conditions should indicate that through email at time of submission of their assignment and will not be considered for the contest.