CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 16
October 29, 2018

Remarks on Upcoming Assignment PS5

Two types of agents

PS4 defined two kinds of agents, *Fickle* and *Follow the Crowd*, but it only implemented Fickle agents.

PS5 allows mixed populations of both kinds of agents.

We do this using polymorphic derivation, introduced in lecture 15.

The polymorphic agent class

`Agent` will become a pure abstract base class.

`Fickle` and `Crowd` will be derived from `Agent`.

`Agent*` pointers will be stored in the agent roster vector. This will allow a mixed population of Fickle and Crowd agents.

Other changes will be made as needed to describe the new more-complicated population of agents.

Remarks on PS4-Consensus

Array versus vector

Q: Should we use an array or a vector to store the list of agents?

A: Use an array as a dynamic extension of the `Simulator` class.

- It's what we've talked about in class. The PS is to give you practice. (See lecture 12, slide 6.)
- It's slightly more efficient.
- You don't need most of the features offered by `vector<>`. Keep it simple when possible.

Composition vs. aggregation

Q: Why can't I just declare my agent array inside the class using `Agent ag[ numAgents ];`

A1: This only works if the value of `numAgents` is known and fixed at compile time. In PS4, it is not known until run time.

A2: Every type has a fixed storage size assigned by the compiler. Composed data members likewise must have fixed size at compile time. Aggregation is the way to model variable-sized real-world objects using C++.

Managing a dynamic extension

Q: How do I create and initialize an array `ag` of `Agent` as a dynamic extension of `Simulator`?

A: Declare a private `Agent` pointer in `Simulator`.
Initialize it in the `Simulator` constructor.
One way uses ctor `ag(new Agent[numAgets])`.

Q: How do I delete the dynamic extension when I'm done with it?

A: Use the destructor `~Simulator() { delete[] ag; }`.

Initializing a dynamic extension

Q: How are the agents initialized when I do

```
new Agent[numAgents]
```

?

A: By the agent's default constructor, which is called automatically for each agent in the array.

Q: I want them initialized using the `Agent(int)` constructor. How can I do this?

A: For each agent `k`, do `ag[k] = Agent(v)`, where the value of `v` is the desired initial choice for `k`. This uses move assignment.

Q: Why not just set the agent's choice to the value of `v`?

A: This would require a setter or other mechanism that violates the privacy of the agent's choice.

Matching `sample.out`

Q: Your `sample.out` lacks the banner and by messages. Aren't we supposed to use them in every program?

A: Yes, you are. Unfortunately, PS4 is a bit inconsistent. It says you should use `banner()` and `bye()` as usual, both of which write to `cout`. It also strongly implies that *only* a single line of numbers should be written to `cout`, and my `sample.out` reinforces that idea.

Q: What should we do then?
A: Comment out `banner()` and `bye()`.

Q: Sample output has extra spaces. Do we need to match that?
A: No.

Running the `sample.in` script

Q: Your code `sh -c sample.in > sample.out` doesn't work for me. Why?

A1: `sample.in` needs to be executable by you. Use `chmod` to fix the permissions.

A2: It won't recognize either `sample.in` nor your `consensus` executable unless "." is in your search path.

Search path

Q: What is my search path?

A: This is a list of directories to search when looking for a requested command. It is a colon-separated list of directories.

Q: Where is it?

It's stored in the environment variable `PATH`. You can see it with `echo $PATH`.

Q: How can I put "." in my search path?

A1: Modify your bash startup file `.bash_profile` where it sets `PATH`.

A2: You can temporarily add it by typing `PATH=.:$PATH` .

Standard Template Class `vector<T>`

vector

vector<T> myvec is something like the C array T myvec[].

The element type T can be any primitive, object, or pointer type.

One big difference is that a vector starts empty (in the default case) and it grows as elements are appended to the end.

Useful functions:

- myvec.push_back(item) appends item to the end.
- myvec.size() returns the number of objects in myvec
- myvec[k] returns the object in myvec with index k (assuming it exists.) Indices run from 0 to size()-1.

Other operations on `vectors`

Other operations include creating an empty vector, inserting, deleting, and copying elements, scanning through the vector, and so forth.

Liberal use is made of operator definitions to make vectors behave as much like other C++ objects as possible.

Vectors implement value semantics, meaning type `T` objects are moved freely within the vectors.

This implies that class `T` should support move constructors and assignment.

Alternatively, one can store pointers in the vector instead.

vector examples

You must #include <vector>.

Elements can be accessed using standard subscript notion.

Inserting at the beginning or middle of a vector takes time $O(n)$.

Example:

```
vector<int> tbl(10);  // creates length 10 vector of int
tbl[5] = 7;           // stores 7 in slot #5
cout << tbl[5];       // prints 7
tbl[10] = 4;          // illegal, but not checked!!!
cout << tbl.at(5);    // prints 7
tbl.at(10) = 4;       // illegal and throws an exception
tbl.push_back(4);     // creates tbl[10] and stores 4
cout << tbl.at(10);   // prints 4
```