Massachusetts Institute of Technology
6.096
Lecture 10
A Case Study – Modeling MIT Students

Overview:

Code has been posted for the simple program demonstration shown in class. The basic idea of the project was to model a set of MIT students, and to display their activities throughout the day. There are obviously different types of MIT student personalities, so we dealt with just a few for the sake of example.

The goal of this case study is not to be a complete or even accurate simulation; it is merely a demonstration of how the concepts we've discussed in the class can be tied together into one large program.

Some of the assumptions and simplifications that were made in developing the model are as follows:

- Any MIT student has a first and last name
- MIT students' most prominently fluctuating features tend to be exhaustion and happiness
- Every student is currently studying calculus, physics, chemistry, biology, and humanities (students don't know how to distinguish humanities subjects), and has a certain level of confidence in his or her understanding of each
- At any point, a student is in a particular state represented by two things: a current activity (which can be sleeping, working, studying, or socializing, as well as playing and practicing a sport for athletes), and a current status message (rather like Facebook status messages)
- Most of a student's study time is spent either on the subject that was just being studied or on the student's least comfortable subject

Class structure:

Two types of students were coded – `NerdyStudents` and `AthleticStudents`. Obviously these are not a representative sample of the spectrum of MIT students, but the two classes are good enough examples for an incomplete simulation such as this one.

In addition to student classes, a `Simulation` class was created. This is a popular technique: if you need to store a bunch of data relevant to the current game or simulation being run, you can create a `Simulation` or `Game` class to store that data and actually run the game or simulation. One `Simulation` object is created per run of the simulation, and the `step()` method is called each time another step of the simulation is to be performed. All `step()` actually does in this implementation is tell each of the student objects to move on to the next activity.

How to use this example:

Most of the code should be self-explanatory, or should be documented with descriptive comments. Look over the code and make sure you understand what it's doing. Make sure you understand why some things are declared const and others are not; why sometimes things are passed by reference and sometimes not; why things are public or protected; when and why pointers, references, and regular variables are used; and so on. If you understand all this, you should be well-prepared for your final project.

The STL

So far the only way we've seen to get a collection of items of the same type is using arrays. However, the C++ Standard Template Library has several classes that make manipulating groups of objects far easier. This will almost certainly be useful in final projects.

STL is a large topic, and we could spend weeks just on using the STL, so don't expect to get used to it all at once.

Before you start learning STL, make sure you remember how templates work with functions. (We covered this in Lab 3.)

Templatized classes:

Just as we can declare function templates, we can also declare class templates. For instance, we could take the `Array` class developed at the end of Lab 6 and make it generic – we could make an `Array` class for use with any type, not just integers:

```
<template class Type>
class Array {
public:
      Type *internalArray;    // Stores a pointer to a dynamically
                              // allocated array (set in constructor)
      Type &getValue(const int n) { return internalArray[n]; }
      // And so on
};
```

To declare an `Array` object using this class, we need to specify a type for the `Type` parameter to take on in the template. We do this as follows:

```
Array<int> nums;  // Creates a new Array of integers
```

STL container types:

The STL provides several kinds of classes for storing groups of data – "container" classes. The `vector` class, from include file `<vector>`, stores elements internally in an array, and can be accessed like an array. `deque` (from `<deque>`) is similar, though implemented slightly more efficiently for some applications, and `list` (from `<list>`) implements a “linked list” – a data structure in which elements are not located together in one block of memory.

The full list can be found on one of the many sites offering STL documentation.

Using STL containers:

To declare a new vector to store integers, we could say:

```
vector<int> nums = {1,2,3};
```

(The vector class has a number of constructors, one of which takes a C-style array.)

We can then use subscripts with this vector as we would with an array:

```
v[2] = 4;   // Sets the 3rd element to 4
```

However, we can also perform dynamic operations on this vector:

```
v.push_back(4);          // Adds the number 4 to the end of the vector
v.insert( v.begin(), 4); // Inserts 4 as the 1st element of the vector
```

For most purposes you’ll be using them for in your project, `deque` and `vector` are probably the only two containers you’d want to use. Using `deque` will likely be somewhat more efficient.

STL iterators:

For many of the data structures in the STL, plain old pointers won’t work properly – we can’t navigate to the next element in a linked list by saying `elementPtr++`, because we don’t know where in memory the next element is. The solution is *iterators*, which are essentially intelligent pointers, implemented as classes. An iterator knows how to find the next element in any kind of STL class, for instance.

Just as we can loop through an array with subscripts, we can loop through a `vector` or another STL container with iterators:

```
for( vector<int>::iterator iter = nums.begin();
     iter != nums.end(); v++) {
      *iter = 0;  // Dereference operator works like with pointers
}
```

The `begin()` and `end()` functions return iterators pointing to the beginning of the `vector` and one past its end, respectively. The mess before the word iterator just indicates what type of container this is an iterator for – in this case, a `vector` of `ints`.

A `const_iterator` is the same thing but without the ability to change the values it points to. In other words, a regular iterator (in our example above) is like an `int *`, while a `const_iterator` is like a `const int *`. It is declared exactly the same way as a regular iterator, but instead of `::iterator`, put `::const_iterator`.

There are many STL functions that take iterators as arguments. For instance, the `insert()` member function of `vector` and `deque` takes an iterator argument indicating where to insert the new element.

It is strongly recommended that you use STL classes instead of arrays. If you do this, though, you will probably need to look up some of the documentation for these classes online. Use the resources listed on the course website.