Assignment 2: STiVo

Ship Date: February 13, 2005, 11:59pm

In this assignment, you'll practice building full-featured C++ classes, using the STL, and writing unit tests. You'll also have to translate written requirements into code and integrate with legacy code. We've also thrown in a design pattern from class for the heck of it.

Some advice up front: read the entire handout before you start coding! Some of the features you'll be implementing in later parts may impact your design decisions.

STiVo: The Stanford PVR

The problem with most PVR's (personal video recorders, like TiVo) is that they aren't custom tailored for on-campus use at Stanford. That's why the Stanford Office of Products We Can Sell To Students (SOPWCSS) has been working on STiVo – the Stanford PVR. The system is being written in C++ (of course) and you've been contracted to help work on the back end.

If you're not familiar with PVR's, here's a quick tutorial. PVR's are like VCR's that use a hard drive instead of tape. They're pretty smart – you can tell them to record every episode of a particular show and you can prioritize shows in case two shows you're recording are on simultaneously. We'll be focusing on some of the underlying pieces of the PVR in this assignment. In particular, we'll be working on some of the underlying data structures and program scheduling classes.

Part 1: The Show Class

The project is in desperate need of a class that represents a single television show. A show has the following properties:

- Name (string)
- Channel (int)
- Start Time (double)
- Duration (double)
- Flags (bool* -- see below)
- Priority (int)
- Description (string)

The Easy Fields

The name, channel, description, and priority fields are all pretty straight forward. The name is just the name of the show. **You can assume that names are unique in STiVo.** In other words, you can determine that two Shows are the same by comparing their names. The channel is where the show is shown, the description is the information about that particular episode, and the priority

is a number that represents how important the show is to the user. A priority of 0 is the default. A higher priority means that the show is more important. Your Show class should have getters for the name, channel, and priority. Setters are not required because the values are assigned in the Show constructor (see below).

Times in STiVo

The start time is represented as a double to make things easier. Every show begins on either an hour or a half hour. The time field is simply an offset (in hours) from a fixed point in time. For example, if "Good Morning, Toyon" begins at time 3.0, that means that it begins at 3 hours after "the beginning of time" (from STiVo's point of view). If its duration is 2.5, that means it is a two and a half hour show and ends at time 5.5. In STiVo, you don't need to worry about times or durations that don't fall on an hour or half hour – it won't happen.

Show Flags

TV shows have *metadata* associated with them. For example, some shows are available with closed captioning. Some shows are simulcast in Klingon (where available). In STiVo, this metadata is represented by *flags*. Each flag is simply a bool that indicates whether the program has the property or not. For example, if a show has closed captioning, the flag corresponding to closed captioning will be true.

The actual meaning of the flags is determined outside the scope of the Show class. All you need to know is whether a particular flag index is true or false – you don't know or care that flag 12 actually means "Recorded in Dolby Digital 8.2 Surround."

Each program has an array of these flags that default to false and can be turned to true by other parts of the STiVo system. Note that flags are runtime settable. In practical terms, that means that when a Show is constructed, you allocate space for the flags, which are all false initially. As the program is running, flags can be activated.

The Show Class Requirements

Various parts of STiVo have already been written with the assumption that the program class will be completed soon. The other components have made assumptions that result in the following requirements for your Show class:

- There must be a no argument constructor that assigns sensible defaults.
- There must be a constructor that takes the name, channel, start time, duration, number of flags, and priority (in that order)
- Show objects will be passed to various functions, so they must be proper C++ objects and behave appropriately when copied or assigned. They should also be const-correct.
- STiVo is designed to run for days on end, with new programs coming into the system and older ones being removed. To avoid crashing, your Show class can't have any memory leaks.

- To comply with the regulations imposed by the National Broadcasters' Association of Arbitrary and Contrived Standards (NBAACS), flags must be internally represented as a heap-allocated array of bools. The array must be the size specified in the constructor.
- The SOPWCSS has had problems with contractors in the past, so they want you to write a suite of unit tests demonstrating that each feature of the Show class works. Your tests should be written in a class called ShowTest with a public static method named "test" and should make use of the CS193D testutils functions.

In addition, your Show class must support the following public methods:

- getName(), getChannel(), getStartTime(), getPriority(), which should all be pretty self-explanatory.
- getEndTime(), which is calculated based on the start time and duration.
- isFlagged(int i), which returns true if the flag at index i has been set. All unset flags are presumed to be false. So if (i >= numFlags), isFlagged() should return false.
- setFlag(int i), which sets the flag at index i to true. If there is no such flag, setFlag() does nothing.
- getDescription() and setDescription(string), which are used to store episode blurbs for the particular showing.
- display(), which renders the details of a Show to stdout. The order and formatting of the details is up to you. Just make sure that you at least output all that data that you have getters for.

Once you have tested your Show class in isolation and you feel that you have the necessary test coverage, move on to the next task…

Part 2: The Schedule

The Schedule class determines what show to record at a given time. Since a STiVo box can only record one show at a time, the Schedule class is to be implemented as a true singleton, as described in class. In other words, it should have a static instance() method that returns the one instance of the Schedule class:

```
static Schedule& instance();
```

There are only two public methods you need to implement for Schedule – addShow() and getShowAtTime():

```
void addShow(const Show& inShow);
bool getShowAtTime(double inTime, Show& outShow) const;
```

The addShow() method

The addShow() method is used by another part of the STiVo system to add a program to the schedule. The idea is that every night, the system will download new information about the shows that will be on. For each show, it will create a Show and call addShow to put it on the schedule. Note that the added program is a *snapshot* of the inShow parameter. You don't need to worry about what happens to the program after it has been added. Also, don't assume that addShow() is called in ascending order of time. Shows can be added in any order.

addShow() is the Schedule's one chance to learn all it needs to know about a particular airing of a program. When addShow() is called, your Schedule will need to somehow remember the details of the Show so that later on, when getShowAtTime() is called, it can select the correct program to record.

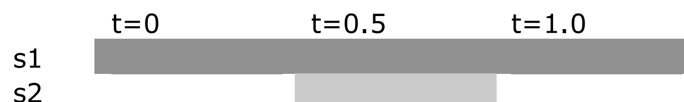
The getShowAtTime() method

getShowAtTime() is called when the system needs to know what show, if any, it should record. The Schedule will need to find the highest priority show at the specified time and fill in outShow with that data. If no show is available to be recorded, it returns false. Otherwise, it returns true.

STiVo doesn't operate quite like other PVRs when it records. Specifically, STiVo can abruptly cut off a show if another with a higher priority is starting. For example, suppose the system constructs the following Shows:

```
Show s1("s1", 13, 0.0, 1.5, 0, 1);
Show s2("s2", 14, 0.5, 0.5, 0, 2);
```

On a chart, the schedule looks like this:



If getShowAtTime() is called at time 0.0, s1 will be selected. If it is called at time 0.5, s2 will be selected because it has a higher priority. At time 1.0, s1 is selected again. The ability to split shows up like this should make things much easier. You don't have to worry about interrupting shows, or remembering what was recorded. You can treat each call to getShowAtTime() in isolation. In fact, each call *should* be isolated. STiVo can call them in any order at any time.

If two shows have the same priority, the Schedule can arbitrarily choose one. In fact, it doesn't even need to stick with that show. If you select show A (starts at time 3.5, duration is 1.0) at time 3.5 and then switch to show B (same priority, starts at time 3.5, duration is 1.5) at time 4.0, that's fine. We won't test this particular edge case, unless you choose to implement "steady shows" in Part 3 (see below).

Storing the Data

The tricky part of the Schedule class is how to store the data. The simplest way of remembering the programs might be to add inShow to a vector each time addShow() is called. We're definitely

not recommending that though. If you went with that approach, every time getShowAtTime() was called, you'd have to loop through every Show to find only the ones that were showing at that time. Then you'd have to find the one with the highest priority. It'd be pretty messy and not very efficient.

There are several ways to store the data that would result in cleaner and more efficient code. We're not looking for any one particular solution, but here are some hints to get you started:

- Look through what's available in the STL container classes, including containers we've talked about in class and those that we haven't.
- If it makes the code cleaner, memory consumption is an acceptable tradeoff. In my solution, I potentially have several copies of each Show because it makes it much easier to do the lookup.[1]
- The time span factor is probably the hardest part of this. Think about what would be the easiest way to find all the programs that are on at a particular time. Now think about what addShow() would have to do to facilitate that.
- We encourage you to talk with your classmates and brainstorm different approaches. The code must be your own, but we're happy to give you feedback at office hours and discuss alternatives.

As before, your Schedule class should also be unit tested thoroughly in a class named ScheduleTest.

Part 3: Feature Creep

Software projects often suffer from a phenomenon known as *feature creep*. It basically means that the longer a software project is in development, the more little extra features tend to find a way in. This can happen for a bunch of reasons. Could be actual external customer requirements that are discovered, could be that a new Product Manager is hired and has her own agenda, or it could be late night coding that results in a cool new feature.

In this case, the SOPWCSS has discovered extra money in their budget and is able to contract you for more coding time than was originally expected. There are two features that they'd like to add, but they can only afford for you to do one of them. So you get to pick – **for part 3, just do one of the following features!**

Feature A: Steady Shows

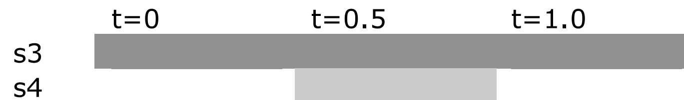
Some beta users have expressed concern over the fact that STiVo can arbitrarily switch between two overlapping shows of the same priority. Given the rules described above, suppose we had two shows constructed as follows:

---

[1] Actually, this is debatable in the real world. When I worked at WebTV, I once tried to add a single char[255] data member to the main Show-like class. It had to be removed because the box dealt with 14 days of programming * 24 hours * 2 shows/hour * 200 channels * 255 bytes = about 32MB! It was a lot at the time, especially since there was a modem involved. For our purposes though, go nuts with memory.

```
Show s3("s3", 13, 0.0, 1.5, 0, 1);
Show s4("s4", 14, 0.5, 0.5, 0, 1);
```

On a chart, the schedule looks like this:



In Part 3, we decided that the show returned by getShowAtTime(0.5) could be either s3 or s4. Since they both have the same priority, either answer was acceptable. Feature A guarantees that the Schedule will stick with s3 the whole time, or until a higher priority show comes along.

There are probably many ways to solve this problem, but the most elegant technique probably involves recursion. If you haven't done recursion before, you might want to either do some research first, or focus your efforts on Feature B.

Note that this feature **only deals with shows of the same priority**. The Schedule will continue to flip between shows if a higher priority shows comes on, as shown in the table in Part 3. Also note that if two programs with the same priority *start* at the same time, you're going to just have to pick one. This feature does not change that particular case.

Feature B: Visual Schedule
Since a flashy user interface for STiVo is still under development, users would like an easy way to view parts of the schedule through the Schedule class. Feature B involves adding the following method to Schedule:

```
void display(double inStart, double inEnd) const;
```

The display() method will render the schedule from time inStart to time inEnd as a human-readable table. The exact formatting of the table isn't especially important (we don't care how many tabs you use) but it should roughly match the following example.

Suppose there were four Shows in the system:

```
Show s5("s5", 13, 0.0, 1.5, 0, 5);
Show s6("s6", 14, 0.5, 1.5, 0, 4);
Show s7("s7", 15, 1.0, 0.5, 0, 6);
Show s8("s8", 16, 2.0, 0.5, 0, 2);
Show s9("s9", 13, 8.0, 1.0, 0, 4);
```

The output of display(0, 2.5) would look something like this:

```
        0     0.5   1     1.5   2     2.5
```

```
s5           +     +     -
s6                 -     -     +
s7                       +
s8                                   +
s9
```

The schedule forms a tab-delimited grid. A blank cell means that the show is not on at that time. A '-' means that the show is on, but won't be recorded. A '+" means that the show is on and *will* be recorded. So you should never have more than one '+' in a single column.

Choose Your Poison
The difficulty of these two features is about the same. Feature B probably involves more code, but Feature A probably involves more planning and debugging. The choice is yours. And yes, you can do both and take a shot at up to 5% extra credit.

Part 4: Integration

Now that you have a working Show class and a working Schedule class, it's time to integrate with the rest of the STiVo system. In theory, if your Show and Schedule classes are written to spec and thoroughly tested, this should be easy. In reality, it may take a bit of futzing with headers and digging into the STiVo code to get everything working.

When you think you're ready, head on over to /usr/class/cs193d/assignments/stivo/ and grab the existing STiVo sources. Drop them into your directory, comment out any existing main() you might already have, fire up gcc, and see what happens. If everything goes well, the STiVo system should launch into its own unit testing suite and all of the tests (including those that make use of your Show and Schedule classes) should pass.

If things don't go well, don't panic. The STiVo code we're giving you is fairly straightforward and you should be able to step through it to see what the problem is. Once all of the tests pass, your code has successfully been integrated into the project. Note, however, that the tests in stivo.cpp are very rudimentary. Your own tests will almost certainly be more thorough.

Getting Started, Grading, etc.

Getting Started
We're not giving you header files this time. Everything you need should be contained in the description above. If your headers don't comply with the guidelines of the system, it should be clear when you get to Part 4.

We didn't include a Makefile with this assignment. You should compile your code manually, like this:

```
g++ *.cpp
```

Assuming your code all compiles and links, you can run the program by typing:

```
./a.out
```

Deliverables
You will need to turn in your Show and Schedule classes, the unit tests for both classes, and a README. The README should briefly describe the data structures and overall approach of your Schedule class, and what tradeoffs, if any, you had to make. You should also specify which feature you implemented in Part 3.

When you're ready to turn in your assignment, type "/usr/class/cs193d/bin/submit" on the elaines and follow the directions.

The following areas will be considered when we grade your assignment:

Correctness. We will run a suite of our own unit tests against your classes, a superset of the tests run in Part 3.

Testing. We will examine your unit tests to ensure that they adequately cover the use cases. You don't need to go nuts with testing. Just make sure that you have a test for each feature and you've covered a number of edge cases.

Scalability. We will inspect your code for memory leaks and make sure that your classes are good C++ citizens (e.g. copy ctors when needed)

Readability. We will look over your code expecting to see appropriate variable and method names, good functional decomposition, and appropriate use of language features.