Programming Project 1†
(Due Thursday, December 25, 2003 at 5:00pm)

About cooperation with other students:
This project is meant to be done alone. Absolutely no cooperation is allowed. If there are
questions, ask the course staff; they are there to help you. It has been observed that
students have coordinated in exercises beyond what was allowed. For this project, you
must do all thinking, all coding by yourself. Make it a point that you will not discuss the
project with anyone else other than the course staff.

Introduction:
Instead of getting depressed by the fact that you got a medal for the most disorganized
person of 2003, you plan to write an Organizer in your new favorite language, C++. Of
course, 2003 was not a good year for you, so you delayed writing that program until mid-
December when you suddenly realized that your 2004 would be as bad as 2003 if you
don’t start on your Organizer idea right away.
While you wanted to have an organized future, you also wanted to excel in your new
favorite language, so you set the following goals for the Organizer program:

1. It should enhance your object-oriented design skills, i.e. breaking large problems
   into objects.
2. It should refresh your coding skills.
3. It should enhance your thoughts on data abstraction.
4. It gives you experience with C++ syntax that you have already started liking.

What you need to do?
Your organizer is a program that runs from the command line in a loop. We will provide
you a starter file to which you can add your code. You will need to write other classes in
different files. We strongly suggest you use one .h file and one .cc file for declaring and
defining each class. The class definition should go in the header file and the methods
should be defined in .cc file.
Your organizer reads in events from the command line as you enter them and adds those
to internal data structures. Run the starter code to get a feel of how the organizer interface
works. The interaction of the user with the program should be robust and error resilient.
For example, entering alphabets for date should make the program scream at you.
For input validation, you will write a class called SimpleInput whose sole responsibility
would be to take the input from the command line and do some validation. You will need
to use I/O capabilities of C++ that will be covered in class but you are encouraged to read
ahead. The SimpleInput class should be able to read a line from the standard input, with

† This project is extracted from an assignment that was created and later modified for use in a Java class at
Stanford University, computer science department.
or without prompt. This is accomplished by providing an overloaded version of ReadLine
method inside the class itself. Similarly, ReadInteger is used for reading integer values,
again with or without prompt. You might want to add methods like ReadDate and
ReadTime to the SimpleInput class. Now comes the good news, we will give you a basic
implementation of the SimputInput class. You are free to modify the SimpleInput class
any way you want. The only thing you need to remember is that the methods provided by
the SimpleInput class are static. What is means is that you can call them directly without
creating an object. For example, if you want to call the ReadLine method of the
SimpleInput class, it can easily be done by writing SimpleInput::ReadLine() somewhere
in your code.

In addition, you will write code for several other classes. And guess what? You are not
left alone; we will help you identify most of the classes and objects to realize the basic
functionality of your organizer.

Requirements:

Here is what is required of the organizer:

- It should present a prompt to the user who can select one of the various options. Run the starter code provided to you to find out about the available options. This requirement has been taken care of by the course staff in a non-professional way to get you started on the real stuff. Just entering the first letter of the command in the command loop should work. This has also been taken care of by us.

- If a user provides an invalid input at a prompt, the prompt is repeated until the user enters valid data. Make sure you read the implementation of SimpleInput class and the SimpleInput.h file to see what some of those methods might return.

- At the prompt(s), user should be able to add an event which the organizer stores internally. Before adding the event to the internal database, it should be verified that the new event does not conflict with any previously scheduled event. If that is the case, a warning is issued telling about the existing conflicts. The event is not added to the internal database and the user is prompted again.

- You need to keep track of the start and end times of an event. We don’t want you to be concerned about seconds, so hours and minutes are just fine.

- User should be able to cancel an event. To accomplish this, user selects the cancel option. The user is prompted with the date, and the user enters the date. Schedule for that date is displayed and the user is asked to enter the event number on that date which is desired to be cancelled. Upon valid input, the event is cancelled and refreshed schedule for that day is displayed again.

- No organizer is useful without some searching capabilities. When the user selects the search option, a prompt is displayed to enter the string to be searched. After user enters the search string, entire database of internally stored events is searched for that string, and all events with their own information are displayed.

- When the user opts to print the schedule, a date is prompted upon entering which the user is able to see the complete schedule for that day, sorting events by their start times. If no event is scheduled on that date, the organizer notifies the user.

- User can choose to quit from the program but making that choice.
Grading:
A good part of the score for this assignment comes from evaluating its functionality which covers your program's behavior from an external perspective. Without looking at the code, does your program work as expected?

In addition to functionality, we also expect that your programs will be cleanly written and easy to understand. Your program should exhibit decent sensible coding practice, such as appropriate algorithm choices, effective decomposition, no duplicated code, as well as things such as appropriate commenting, well-chosen identifiers, consistent indentation and capitalization, and most importantly, good object encapsulation. If your program fails to work on any of the functionality tests and we go hunting in your source code to understand the problem, how easy your code is to understand will have a large effect on our ability to give partial credit.

Where do you start?
A good starting point will be to run the starter code provided to you through the class website. Go through the interactive portion of the program. Then open up the files we have provided and try to understand the bigger picture. For this project, we are providing you most, perhaps all, of the classes that you will need. Feel free to add as many classes as you want. However, before defining a class, think whether you need that class or not.

Since this is the first big object-oriented code that you will write, we will give you some useful guidelines. We describe how you break down the problem into objects and mention some of the milestones that you should check while writing your code.

Designing your classes:
An object-oriented program is decomposed into a number of cooperating classes. To define a class, you need to answer two fundamental questions:

- What data is required to represent an object of this class?
- What behavior will I need from that object?

The answers to these questions tell you what data members you need and what methods the class must implement. Behavior that is associated with an object should be provided as a message you can send to the object to ask it to perform some action, rather than having the client reach into the object and muck around with its data. For example, a client asks a Time object to find out whether it is before another Time object. If you want an Event printed, you send a message to that Event object asking to print itself.

To introduce you gently to this process, only a few simple classes are needed for this project. These objects include the Time, Date, Event, TimeInterval, DailySchedule, and Organizer classes, along with a few helper-utility classes and routines. Each of the main classes has a pretty clear real-world analog to help guide your design and the relationships between the classes are not too complex. Let's give you an overview of each of the classes so you'll know how to proceed.
The Time Class

First consider the Time class we have made available to you through the resources section of the class web page. It has the straightforward job of representing a particular time in the day and printing it. This is a primitive version of this class, and you’ll need to further develop it to be fully useful.

The Time class is your first opportunity to work through designing a useful and robust object with a sensible and complete interface. Carefully think through your decisions and make choices of which you can be proud. It's not a very complex class, so it is a good one to tackle early, so you have a simple introduction to designing and manipulating data in the object-oriented paradigm.

First, evaluate the current interface of the Time class. Is it missing important features? Does it have unnecessary functionality or data members you want to remove? Think through the rest of the program and what operations you will need to manipulate times and plan to include them as methods for the Time class. Feel free to remove any current methods that are redundant or unnecessary in your design. Be sure to only make those things public which make sense as part of the Time's external interface and which you are committed to supporting forever.

This Time object has the hour/minute/am representation that is intuitive and familiar, but it can be awkward to manipulate in that form. What other representations might work? What are the tradeoffs in terms of space and convenience of these other choices? Consider what operations are easier for a given representation and which ones are more difficult. Think through these before committing on your representation and then go forth and complete the implementation.

Instead of moving on, write some simple test code that exercises the Time class and allows you to find and correct any problems now. Can you reliably create Times, message them, print them, compare them, shift them, etc. using your operations and get the correct results? It's much easier to isolate and fix bugs when you're just dealing with one class at a time than trying to sort out everything at once when you have everything lumped together into a bunch of untested classes. One convenient place to put your test code is in MyFirstProgram.cc file from exercise 1.

The Date Class

We suggest you deal with this class next. The Date class is very similar to the Time class and same thought process should be applied here as was applied in the case of Time class. Before moving on, write some test code to test the Date class.

The Event Class

Now consider the Event class. An Event object encapsulates all of the details for a particular organizer entry. An Event needs to track its name, location, an optional comment, and some information about when it happens. Some of these fields can easily be represented using strings. In truth, the Event class is not that much more sophisticated than a struct, but it serves to encapsulate the data into a clean abstract unit. A good milestone to aim for before moving on from Event is that you are able to read the events from the keyboard and print them out to verify that the Event class is doing its job. To do so you might need to implement TimeInterval class first.
The TimeInterval Class

Managing the time interval for an Event is a little more complex than the other simple string fields of the event. You may want to create a helper class, TimeInterval, just to encapsulate this concept. You may choose not to have this class at all if you can deal with the duration of an event in some other way. Remember, whatever way you use, an event should itself be responsible for maintaining its duration. It is certainly possible to just directly track the start time and duration in the Event object as an alternative.

Internally, a TimeInterval could store its data as an array of two Times, one start, one stop, or perhaps a starting Time object and duration, or perhaps something else entirely. Wrestling with these sorts of decisions is the most interesting part of designing a class. As a client of the TimeInterval, you don't care how it is internally represented; all you care about is that you can get the right results when you ask it to print and whether it starts before or overlaps with another TimeInterval and so on. But as the implementor, how you choose to represent it can make quite a difference in terms of ease of writing the code, the resulting efficiency, the size of the object, how difficult it is to modify later, etc. In general, we encourage clean design and clearly written code that may be less efficient over the terse, complicated alternatives that are produced in the name of efficiency.

The DailySchedule Class

To manage all the events scheduled on a particular day, you will need to create the DailySchedule class. Since the number of events is not known in advance, a static array is not appropriate for storing Events inside the DailySchedule. Instead, you should use a way where Events can be dynamically added to the DailySchedules. There are two ways of doing it: the hard way is to use an Event pointer inside DailySchedule and allocate memory for the Events as they come before adding them to the DailySchedule, and the easier way is to use a built-in helper class. It turns out that vector class provided by the standard C++ library would do the job the easier way. If you chose to use a pointer, also take care of deallocating the memory you allocated before the program quits. While standard C++ vector class is one easy way to hold events in a DailySchedule, also think of other alternatives.

One bit of suggestion: in order to facilitate orderly printing of the daily schedule, it is convenient to keep the vector of events sorted by start time. The DailySchedule class will include functionality to add events and print out the schedule in various formats.

The Organizer Class

Finally, you will create an Organizer class that tracks the schedules for all the various days. You need to be able to lookup a particular DailySchedule given its Date, as per program requirements. Carefully think about the alternatives you have for defining the data members and the methods of this class. Think about what happens when the user tries to look up the DailySchedule for a Date for which no event has been scheduled.

The Organizer class serves as the database of every piece of information needed to accomplish different tasks at the client level. We suggest you add functionality to this class in the end once you have a grasp on how to implement classes.
AppointmentBook Class

This class is designed to run the interactive portion of the program. It should create an Organizer within itself. We are providing you partial implementation of this class. You need to add functionality to this class to make it work as desired. To get a feel of how the project will turn out, compile and run this class. It also contains a main method which you can also use to test your other classes. As already mentioned, you can also use other programs, for example MyFirstProgram.cc, to test your class files.

What and Where to submit?

1. Some people are still not sticking to the guidelines. Very importantly, strictly stick to these guidelines.
2. The assignment has to be done in Unix environment.
3. When you are ready to submit, place all your .h and .cc files in one folder on a windows machine in the lab. Zip all files together in one zip file whose name should be <your_student_num>.zip where <your_student_num> is an 8-digit number. Be careful with this, our script might throw away zip files that do not follow this convention.
4. Do just one submission of your zip file by dragging and dropping it in the following folder over the network:
   \badar\Common\cs292\section1\project1
   OR
   \badar\Common\cs292\section2\project1
   depending upon your section. DO NOT submit in the wrong section. Note that you will not be able to open the project1 folder. Just drag and drop your zip file in there.