Project 2
A Spelling Checker Using Basic C++ Concepts:
A Templated Container with Iterator Interface and a String Class

Due: 11:59 PM Friday, Feb, 13, 2008

Note: The Corrections & Clarifications posted on the project web site are officially part of these project specifications. You should check them at least at the beginning of each work session.

Purpose

In this project, you will practice the following:

- Programming classes using a specification of their public interface and behavior that work as fully reusable components.
- Programming user-defined types with overloaded operators to permit their being used as easily and flexibly as a built-in type
- Programming classes that manage dynamically allocated memory with constructors, destructors, and assignment operators.
- Using static member variables and member functions to work with "class-wide" information.
- Using exceptions for error handling.
- Using C++ console and file I/O

This project will benefit from the work you did in Project 1 in that basically, it is a "clone" of Project 1 but uses the facilities of C++ to greatly simplify the main body of the code through two classes:The *String* class will automatically manage the memory for strings, and provide a true "string variable" capability that makes the main project code simpler and easier to get correct. Its internals are similar to Project 1's `Ordered_array`. An *Ordered_array*<> class template will replace the `Ordered_array` opaque type to get amore general-purpose component similar in concept to the Standard Library container templates. Instead of a container of `void *` pointers whose usage involves careful casting, the `Ordered_array`<> template can be instantiated for objects of any type, allowing you to store `Strings` and `Action_items` directly in the array cells inside the `Ordered_array`. The container also has an iterator interface, meaning that the client code does not need any information about how `Ordered_array` is implemented internally. In fact, `Ordered_array` could be implemented completely differently (e.g. with a binary tree) and the client code would be completely unchanged (except of course for the difference in memory usage).

You can start with your Project 1 solution and bring up this second version of the spell-checker program by examining your code structure from Project 1 and rewriting it to be clearer and better organized, and conform to C++ idioms and programming practices, such as declaring variables at the point where they can be usefully initialized. Check your code against the C++ Coding Standards on the main web page. You should also take full advantage of the possibilities that the `String` class, with its true variable-like behavior, provides compared to messy hacking around with C-strings. Note especially how `String` has some member functions that are directly useful in the spell-checker project. You must use these instead of low-level coding whenever they will work.

The course web site has handouts that explain the important points about C++ I/O streams and file streams which were assigned reading. Please review these before starting the project.

Program Behavior

The basic behavior of the Project 2 program is identical to the Project 1 program as specified in the Project 1 document and its Corrections & Clarifications. However, any specifications of program behavior in this document and the Project 2 Corrections and Clarifications take precedence over those of Project 1. The major change in the program's behavior is very simple:

No Length Restrictions.

In Project 2, a `String` class will be used for all input, and the input functions associated with `String` automatically expand the internal memory as required. So there is no longer any need to restrict the length of the input lines, words, or file names. Consequently, the "length problems" demonstrated in the Project 1 samples no longer present any problems for Project 2. See the posted samples for a demonstration.

However, the output of the memory allocation command will be somewhat different due to the use of a templated container for Ordered_array.

Required Components

This section describes the required components that you have to implement according to specifications supplied here. These components are specified in terms of the files that you will submit to the autograder system. You must supply all of these files, and no others. Where noted, the components have "skeleton" header files available from the project web page or file server directory that you must turn into complete header files. These skeleton header files contain specification details that are part of the project specifications, and should be examined with care.

The header files specify the public interface of the components. Thus you may not add, remove, or modify the public members of any classes in the skeleton header file, or add or remove any prototypes for any non-member functions in these header files. You also may not change, remove, or add any friend declarations - these are also part of the interface for a class. Our solution should be able to use your components, and your solution should be able to use our components. This will only work if your components have the same public interface and behavior as our components. If we specify a private member, it is because it is critical that you implement the class in terms of this private member, either for our testing purposes, to guide you to a good solution, or to help ensure that you learn important concepts.

String.h, .cpp

Study the posted demonstrations to see how `Strings` can be used, and to see detailed examples of `String`'s behavior. The supplied skeleton header file defines the public interface of this component, These are the public members of the `String` class, and some non-member functions that are associated with the `String` class, such as the input and output operators. *You may not change, add, or remove public class members, nor may you change, add to, or remove any of the specified non-member functions*. Please ask for help if you think you need to violate this restriction in order to complete the project. You get to choose the private members functions of the `String` class, which are"helper" member functions, but only these private members are up to you. You can have additional non-member functions of your choice in the .cpp file as long as they have internal linkage. Your implementation must conform to the specified behavior in the header file and the project documents, and illustrated by the demonstration code, but otherwise, how it works internally is up to you. See the suggestions below for how to develop this class relatively painlessly - many of the functions can be implemented easily in terms of a few "building block" functions, and several aspects of `String`'s implementation will be familiar from Project 1's `Ordered_array` module.

The `String` class is specified to include some static members that count the number of `String` objects in existence, and the total number of bytes allocated by `String` objects.

Also, the `String` class public constructors, destructor, and assignment operators emit certain messages to make it easy to tell when they are called. These facilities are not part of a normal implementation, but provide an easy way to monitor how the `String` objects are behaving. The messages include the string data to make it easier to tell which `String` object is involved. Note that only these specified public member functions emit the messages directly; other member functions may result in the messages because they call these member functions in order to do their work. To avoid fatal annoyance, the messages can be turned on or off with an additional static member. See the skeleton header file for details.

Ordered_array.h

Study the posted demonstrations to see how the `Ordered_array` container class template can be used, and to see detailed examples of its behavior. The supplied skeleton header file defines the public interface of this component. Like the `String` class, you may not modify the public interface, but the private members are up to you (with an exception for the `Iterator` class). Internally, the `Ordered_array` class behaves the same way the Project 1 module did; you should be able to recycle much of the code. Note that there is no .cpp file for this component because all of a template's code must be contained in the header file for the compilers we are using. However, if you follow the advice on the easy way to develop a template, you can have a .cpp file for this component while you are developing it; but your final component is only a header file. In addition to this component being a C++ class, there are two major differences between this component and the Project 1 component:

First, the `Ordered_array` class does not store pointers to objects, but ***actual*** objects are stored in the internal array directly. In other words, `Ordered_array<int>` contains an array of `int`s, `Ordered_array<Action_item>` contains an array of `Action_items`, and `Ordered_array<String>` contains an array of `Strings`. The reason why the Project 1 container stored `void *` pointers in the array is because that is the closest you can get in C to having a generic container that will hold any kind of item. C++ templates allow us to store items of any type directly in the container. The comparison function will thus receive actual objects to compare, rather than pointers to objects. Of course, one can store pointers in this `Ordered_array` container, but for purposes of this project, the actual `Strings` (for words) and `Action_items` must be stored in the `Ordered_array` containers. Since the container will hold a copy of the supplied item, and will have to move the items around in the internal array and copy them when items are inserted and the array is reallocated, any objects stored in an `Ordered_array` container must have a public default constructor, destructor, copy constructor, and assignment operator. These must work correctly, or else a mess will result when the objects are put into an `Ordered_array`.

Second, instead of using array subscript values, `Ordered_array` communicates with the client code through *iterators*. Iterators allow the client code to "point" to items in the container without have to declare or otherwise "know" how the items in the container are organized or referred to. The iterator interface is very similar to the Standard Library containers, so its usage may be familiar: iterators behave analogously to pointers. Basically, the `find` function will return an `Iterator` object that "points" to the found item in the container. If you dereference the `Iterator`, you get the pointed-to item (a reference to it). If you use the arrow operator with the `Iterator`, you can access a member of the pointed-to item. If you increment the `Iterator`, you point to the next item in the container. If you want to remove an item, you supply the `Iterator` that points to it. See the posted demos for examples of how `Ordered_array` is supposed to be used.

Action_item.h, .cpp
See the supplied skeleton header file. `Action_item` will be a struct that provides the same information as the Project 1 did. However it contains two `Strings` for the match and replace strings. It also has constructors to simplify creation of `Action_items`, and an overloaded output operator to simplify output.

Utilities.h, .cpp
As a bit more template practice, `Utilities.h` must contain a function template named `swapem` that exchanges the values of two arguments of the same type. See the skeleton header for details. Your code should use this template wherever appropriate to do so. You can also include in this file other functions of your choice following the same rules as stated for Project 1. Your `Utilities.h`, `.cpp` will be included in any builds that use your code, so you can plan on using the contents in all of your code. To keep the autograder happy, your submitted project must include `Utilities.cpp` even if it is empty and does nothing.

These files are for handy utility functions that good programmers develop and keep around to simplify their programming. In addition to the `swapem` template, you can put anything you want in this module, but you should include functions that are either used in multiple .cpp files, or might be used in other projects in the future. Functions that are used only from one .cpp file, or are completely project-specific, are not good choices. Review the discussion of this module in Project 1.

p2.cpp
This source file contains function `main` and all of the other declarations and functions for the project. Your `main` function should be very small, doing nothing more than setting things up and calling other functions to do the actual work. Each function should be no more than about a page long, and usually a lot shorter. Declare all the functions with function prototypes near the beginning of the file, and put the functions in a comprehensible and easy-to-read order; put main first, followed by the functions it calls, followed by the functions they call, etc. It should be possible to read your program like a story, from the top down, and each function should be easy to find in the file.

Function `main()` must declare/define an `Ordered_array` for the dictionary and another for the action list, and do so before starting the command-handling loop.

Detailed Specifications

Specific requirements

1. The only dynamic memory allocation should be inside the `String` and `Ordered_array` classes. It is not needed anywhere else in the program. Seek help if you think you need it elsewhere.
2. There should be no declared built-in arrays (e.g. `char buffer[127];`) in your program, and the only place dynamically allocated memory can be used as an array is inside the `String` and `Ordered_array` classes. Declaring C99-style variable-length arrays is also disallowed. `Ordered_array` and `String` take care of all of the purposes that you might have needed an array for. Failure to use them where appropriate is a serious conceptual failure.
3. There must not any hard-coded limits on the length of words, filenames, or input lines. The only limit is available memory. Thus all variable-length input should be done into `String` variables - that is what they are for. Presence of limits like those used in Project 1 will be considered a major design failure.
4. The top level of your program should catch any `bad_alloc` or `String_exceptions` thrown by lower-level functions and terminate after some message of your choice.. `Ordered_array` does not throw any exceptions of its own (although it is conceivable that new might throw a `bad_alloc` exception, and some other library function might throw an exception). Since the `String` exceptions indicate programming errors, in a correctly working spell-checker program they should not get thrown either in the samples or in our program behavior testing, nor should any other exception get thrown. However, during our testing of your `String` component, we will test situations in which a `String_exception` should get thrown.
5. Do not violate the restrictions on the public interface of the components. You absolutely may not change these. Where unspecified, you are free to implement the private internals of the class in any way you choose that is consistent with the concepts, techniques, and recommendations presented in this course.

6. Your implementation of the `Ordered_array` and `String` components should create temporary `Ordered_array` or `String` objects only where specified. This is a matter of both reasonable efficiency and to ensure consistency in our testing.
7. The top-level of your program, function `main()`, that reads the commands, should not create any `String` objects. This will prevent the output of the memory allocation command from being distorted by any local `String` variables. Note that the commands, being single characters, should be read into a char variable instead of a `String`. Following the principle of putting variables into the smallest reasonable scope, any `String` variables used for input should be declared in the separate functions for each command (or their subfunctions).

Use of the Standard Library

1. You may not use the Standard Library `<string>` class or any of the Standard Library (or "STL") containers in this project. The idea is to learn how these things are done by writing some simplified versions yourself. Subsequent projects will actually *require* full and appropriate use of the Standard Library.
2. You should `#include <new>` at the appropriate point to access the declaration of the Standard Library `bad_alloc` exception class
3. You should use any functions in the C++ Standard Library (and the included C Standard Library) for dealing with characters and C-strings, including any of those in `<cctype>`, or `<cstring>`. See a Standard Library reference, or the online brief references pages. Any non-standard functions must be written by yourself. Hint: The project can be coded easily using only basic functions like `tolower`, `strcmp`, `strcpy`, `strlen`, and `strncpy`; you may want to rethink your plans if you think you need some of the more exotic library functions.
4. You should `#include <cassert>` and use the `assert` macro where appropriate to help detect programming errors.
5. Only C++ style I/O is allowed - you may not use any C I/O functions in this project. This means you can and should be using `<iostream>` and `<fstream>`. However, since all input of string data has to be done into `String` objects, which are then used in the processing, you will use very few of the possible input and output operators and functions in the Standard C++ Library.
6. You may use only `new` and `delete` - `malloc` and `free` are not allowed.

General Requirements

To practice the concepts and techniques in the course, your project must be programmed as follows:

1. The program must be coded only in Standard C++.
2. You must follow the recommendations presented in this course for using the `std` namespace. Review the Handout. Failure to follow these recommendations will be considered a serious failure
3. The program should be as "bullet proof" as possible and should run without crashing regardless of the contents or length of the input document or gibberish typed in response to commands.
4. You must close each file shortly after the last input or output operation on it.
5. When your program terminates, all dynamically allocated memory must be deallocated - in this project, correctly working destructor functions will do this automatically.
6. There must be no memory leaks.
7. You may not use any global variables anywhere for any reason, and they are not necessary. The Standard Library global objects `cout` and `cin` are the only exception. Violation of this restriction will be considered a serious design failure.

How to Build this Project Easily

Both `Ordered_array` and `String` can be built and tested separately. Do so; absolutely, positively, do not attempt to do anything with p2.cpp until you have completely implemented and thoroughly(!) tested these two components.

The `String` class has many member functions. However, most of these can be easily implemented in terms of a couple of key other functions, some would be private helper functions, other would be public member functions. Remember the key advice: If you factor out common operations into separate small functions, then you only have to get that code correct once and in one place. *Hint*: After coding and testing the "big three", write the += concatenation operator functions. Some of the more complex remaining functions can be written almost trivially easily in terms of +=. For other functions, note that you would have solved similar problems in Project 1.

Also, check out the `istream peek()` member function - this can greatly simplify the `String` input operator.

A common beginner's error with classes like `Ordered_array` and `String` is to not take advantage of the fact that member functions have complete access to the "guts" of the object. Instead beginners try to implement member functions only through the public interface; sometimes the results are extraordinarily clumsy and round-about. By all means call a public member function if it does exactly what you want. But do not hesitate to directly go to the private member variables in writing a member function - there is no virtue in sticking to only the public interface in implementing member functions!

In particular, notice that the `Iterators` in `Ordered_array` are intended for the clients of the public interface - there is absolutely no need to use them in member function code. For example, trying to write `Ordered_array::find()` using only `Iterators` to access the data would be silly (and maybe impossible).

Submitting Your Project

Watch for an announcement that the autograder is open. Instructions and suggestions will be on the course and project web page. We plan to test your `Ordered_array` and `String` components separately, and combine your components with our `p2.cpp` and vice versa to assess whether you have implemented and used the components correctly. Since the spell-checker may not naturally use all of the capabilities of the two components, your component testing needs to cover every function and the range of possibilities.