CSI 333 – Programming at the Hardware/Software Interface – Fall 2000

Programming Project 4

WARNING: Read and follow the instructions herein about Revision Logs before you start this project. Otherwise, you might get ZERO credit or face more serious consequences.

1 Objectives:

1. Practice designing software logically organized into modules.

2. Practice implementing the logical organization by (a) coding each separate module interface in a separate header file, (b) each separate module implementation in a separate .cpp file, and (c) using #include’s to guarantee that interface declarations seen by both the implementation and users of each module are consistent.

3. Develop a string handling and lexicographic comparison modules specialized for the application requirement that memory space is to be conserved.

4. Introduce a precise description of some C/C++ lexical elements to understand the project requirement to scan C/C++ code. Required reading: HS 2.1 through 2.1.3, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7.3, 2.7.4, 2.7.5, 2.8.2.

5. Implement a table driven finite state scanner and program it to do easy analysis in terms of the syntax of C/C++ lexical elements.

6. Practice the implementation and use of a sorting algorithm.

7. Write a “Makefile” and use make to automate safe and efficient software rebuilds.

8. Observe the operation of software on “realistically” sized input files.

2 Advice

This project is designed to be done after you did the modularization and Makefile practice of Lab Exercise 3. Therefore, if you didn’t have your lab session yet, you should work on that Lab Exercise before putting much effort into this project. That Lab Exercise includes starting work on two of the modules required for this project. Notice that a simple strread module implementation code was given out in a lecture.

Before you attempt to write even one line of code, you must read and and analyze the problem statement (“specification”) to understand what your program is supposed to do, and the design and implementation plans to understand how to go about creating the program. Read the whole assignment and submit any questions to the lecture, the newsgroup (sunya.classes.csi333), a TA or the professor.
3 Problem

A cross-reference or index is a sorted report of selected words or other items from a document that shows for each word or item where in the document it appears. See the back of any text or reference book for an example of an index. Note that it omits references for most general purpose words.

This project is to produce a cross-reference for all the appearances of identifiers in a C/C++ program file. You can imagine how useful such a cross reference is to a programmer who is tracking down a bug involving a particular variable or function. An identifier is a name, which could be the name of a function, type or class, or variable.

Other forms of cross-references are the key word in context index (also called KWIC index or permuted index) and a concordance. A KWIC index lists lines of text in which each line is listed several times, once for each (significant) in the line, alphabetized by the word appearance. A concordance lists for each word, all the sentences containing the word and where they occur, usually in a piece of literature.

Tools available in the ECL and most other Unix environments to build an index to rapidly find where each named object is defined in a set of source files are ctags and etags. Read the man pages for them in the ECL. They can be very useful when you are working with a large program.

4 Some Details

4.1 Line Processing

1. For minimum conformance with ISO C, valid input lines will be at most 509 characters including the newline. Your program should handle an excessively long input line by printing the message

   Input line too long...exiting.

   and then exiting immediately. Code to do this has been provided in one of the lectures.

2. The C/C++ lexical notion of line should be used to determine the line number printed in the cross reference report. L The end of a line is marked by the newline character. See HS (2.1.2).

3. (Refinement) A C/C++ logical line ends with a newline character that is not immediately preceded by a backslash “\”. Ignore trigraphs for this project. One logical line is usually comprised of a physical line. However, if a physical line ends with a “\” just before the newline, then the logical line is continued onto the next physical line. Thus the perversely written logical line

   confusion = con\n   fusion + more_confusion ;

   is logically equivalent to

   confusion = confusion + more_confusion;

The cross reference output should be:
confusion
1:3 1:15

more_confusion
2:10

4. String literals (enclosed by matching non-escaped "s) and character literals (enclosed by matching non-escaped \') do not contain identifier appearances.

5. (Refinement) It is somewhat subtle to design the scanner to detect the end of these literals since they can contain escape sequences. You will get most of the project credit for handling string literals if your scanner handles string literals without escape sequences and inputs with no difference between logical and physical lines.

   Again, ignore the ISO C “trigraphs”: Assume an escape sequence is \ followed by any character, except when the \ is at the end of one physical line to continue the logical line onto the next physical line. A further refinement is to handle all the ANSI C escape sequences of HS (2.7.5).

6. C/C++ logical lines and comments, both the C style /* ... */ form and the C++ style // ... (to end of logical line), should be used to determine the text regions in which to find identifier appearances. See HS (2.2) and (2.8.2).

   A /*... style comment begins with a sequence /* that does not occur in a comment, a string literal or a character literal and continues up to the next */ sequence.

   A //... style comment begins with a sequence // that does not occur in a comment, a string literal or character literal and continues up to next end of a (logical) line.

7. (Refinement) Handle continuation lines within C++ comments.

8. Interpret a logical line whose first non-whitespace character is "#" as a preprocessor command. The preprocessor command continues to the end of the logical line.

   Do not interpret any part of the preprocessor command as an identifier appearance. (Basically, skip them except for the line number counting.)

9. (Refinement) Count the one or more physical lines comprising the preprocessor command to determine subsequent line numbers.

4.2 Identifier Appearance Detection

Since this is only a CS3 project, a simplified view of the C/C++ grammar will be taken. We will call a identifier a

1. maximal, contiguous substring
2. of letters, digits or the underscore characters
3. that begins with a letter or underscore and
4. that is not immediately proceeded by a letter, digit or underscore.
(Refinement) Note that identifiers are determined after continuations of lines are processed and comments deemed replaced by whitespace.

A identifier appearance in a line is a sequence of characters that begins at a particular place in the line, is an identifier and is not one of the C/C++ reserved words.

4.3 Excluding C/C++ reserved words

C/C++ reserved words should not be cross-referenced. (To implement this requirement, your program must use a table of those reserved words to tell which words to exclude. I'll provide a text file of them under ~cs1333/Project4 so you don't need to waste time at the keyboard.)

4.4 Lexicographic Comparison

The cross-reference output of identifier appearances will be sorted according to an unusual lexicographic order that makes words that differ only in the case (CAPITAL or lower) of some letters appear adjacent in the output. This helps people find mistakes. It requires you to implement your own function to do the lexicographic comparison for the sorting.

The ordering rules for characters:

1. The underscore precedes all the digits and letters.
2. The decimal digits “0, 1, ..., 9” precede the letters. The digits are ordered in order of their ASCII codes, which is their normal numeric order.
3. Different letters (ignoring case) are ordered alphabetically.
4. The upper case version of a letter immediately precedes the lower case version of the same letter.

In other words, the alphabetizing sequence is “_, 0, 1, ..., 8, 9, A, a, B, b, ..., X, x, Y, y, Z, z”

General lexicographic comparison algorithm:

1. If one of the words is empty, then the empty word is before or equal to (LEQ) the other word.
2. Compare first characters of the two words. If they are different, the lexicographic order of the words is the order of these characters according to this project's character ordering rules.
3. If the first characters are the same, the lexicographic order of the words is the lexicographic order of their suffixes after the first character.

You can implement this either iteratively or recursively.

5 Output

Report the identifiers in the order described above. Each report consists of the identifier by itself on the first line. On the second line, each identifier appearance is reported by the decimal line number, a colon, and the decimal position of the appearance's first character. The appearance
reports must be in lexicographic order of line number followed by position. Adjacent appearance reports must be separated by single spaces.

Each identifier report is separated from the next by an empty line.

6 Example

Input:

```c
int main( int argc, char *argv[])
{
    int sav_argc = argc; //save argc
    cout << "Hello. #arguments=" << argc << endl;
    while( argc-- )
    {
        cout << argV++ << endl;
    }
    //Whoops, I misspelled argv as argV!
    for( argc = sav_argc; argc; argc-- )
        cout << argv++ << endl; //Now I got argv right!!
    exit(0);
    return 0; /*Silly operation never to be done after exit(0)
        Please don’t write SW like this.*/
    argc = argv[argV]bad_c_syntax_.zz*"argc in a literal\n    which continues +argc on another physical line"+argc ;
}
```

Output:¹

```
_ZZ
15:33

argc

argv
7:14 15:15

argv
1:27 11:14 15:10

bad_c_syntax
15:20

cout
4:3 7:6 11:6
```

¹I did this by hand so it might have mistakes.
7 Non-functional Requirement

A non-functional requirement for software is a requirement that does not involve the input-output or "logical" behavior, but may apply to the software's internal structure, implementation language, performance (running speed), memory use, etc. For us, finite state machine scanning to find the word appearances is required for educational purposes. Also, the data structures and algorithms for storing and sorting the word appearances should be provide good performance and memory use even for somewhat large input files.

7.1 Finding Identifier Appearances

General principles of table driven scanners in terms of a small, finite number of states will be covered in the next few lectures.

7.2 Managing Identifiers and their Appearances

A non-functional requirement for this project is that memory use must be minimized so that 333cxref can operate on somewhat large input files, such as source files under various /opt/applications/solaris7.0 subdirectories. The definition of word appearances also makes the software useful bodies of non-C/C++ text. To achieve this, once a line is read its identifier appearances are found and recorded in the database, the storage used to read the line is reused.

The same identifier will typically appear MANY times within the input file. Only one copy of the identifier itself (in a subarray of characters) should be stored in memory.

To achieve that, all identifiers will be referenced by 16 bit index into the string table array where the identifier’s characters will be found. The identifier should be null terminated.

An identifier appearance can be represented by a triple \((l, p, w)\) of indexes: \(l\) is the (physical) line number, \(p\) is the position on the line of the first character of the word appearance, and \(w\) is the index into the string table where the null terminated C-string holding the identifier’s characters.

Sorting algorithms and data structures for identifier appearances and sets of them to which sorting algorithms can be applied will be covered in lectures.

The string table should be implemented by a dynamically extendable array. (First, allocate a free-store array of char with some initial size of your choice. Keep track of how much of the array is used and the index of the next unused entry. If the array is “used up”, allocate another of twice the size, copy the data from the old array to the new one, free the old array’s space and then assign
the address of the new array to the pointer variable used to address the string table. An analysis of this data structure's resource usage will be given in a lecture.)

It is part of your assignment to decide what data structures and algorithms to use to maintain the database of identifier appearances so that new ones can be entered efficiently and so they are reported in lexicographic order. Notice that the line number/position pairs are generated in the same order as they should be reported. The major alternatives to consider are:

1. Maintain the identifiers in a sorted, balanced tree data structure that has good insertion performance, such as a 2-3 tree.

2. Maintain the words in a data structure with excellent insertion performance such as a hash table. Then, sort them after the processing of the input is finished.

8 RCS Version Control, Logging and Submission Checklist

You must submit your program as a set of RCS database files, one for each source file, including the Makefile. (The RCS database files are all named with the source file name followed by the ,v suffix.) The RCS database files must contain your revision log, which we will read using the “rlog *,v” command.

The revision log should show your progress as you develop the software incrementally. Specifically, a log entry should be made at least each time you complete the testing of one function after possibly debugging it. Make additional entries for every design change or design decision for an interface, data structure, or algorithm, or other change in response to finding a bug, reaching a clarification of the specification, or adding a specified refinement.

A submission without a revision log will receive ZERO credit, and logs found to be faked will be treated as evidence of CHEATING.

A word to the wise: Run make and do a final test immediately before submitting your work using turnin. If you edit anything, DO NOT TURN IT IN until after you remake the software and test it again. The remake will verify your edits (or attempts to restore old code, etc) did not introduce syntax errors.

IF WE CANNOT BUILD YOUR SOFTWARE from sources only (no objects or executables) BECAUSE OF SYNTAX OR UNDEFINED SYMBOL ERRORS, YOU WILL GET NO CREDIT FOR PROGRAM FUNCTIONS

9 Test Cases, Grading etc.

1. Look for sample inputs and outputs in ~csi333/Project4/Testing after they are announced in class and on the class web page.

2. The due time is Thursday, November 16, 9:00PM.

3. Late submissions will be accepted until Wednesday, November 22, 9:00PM (6.000 days). However, a lateness penalty factor equal to \((12.0 - ND)/12.0\) where \(ND\) is the amount of time late, measured in days, will be multiplied into your score, computed using floating point arithmetic. \((8.333\% \text{ per day})\) This means the amount deducted for lateness will begin at 0% and rise (almost) continuously to 50% during the late turnin period. Early submissions will be accepted and will earn the usual 5% bonus.
4. 70% of the score will come from evaluation of outputs on test cases. To get any of these points, your software must build to an executable named 333cxref when we run “make 333cxref” in a directory with all the .cpp and .h files that you had submitted. All object (.o) files and executable files will be deleted before this is done; and points will be deducted if any such files were submitted. Give it a final test before you submit it; do NOT open it with an editor AT ALL after the final test. Most of this credit will come basic cases; the “refinements” marked above will amount to a total of 10 – 15%.

5. 15% of the score will come from your report of systematic testing, including regression testing that you should do while you develop the project. Submit a text file named “mytesting.txt” that briefly describes the test cases and procedures (scripts) you developed, together with all the test case input, output and script files. Each test case input should be in a file with a name matching t*.in; the corresponding output file would be named by replacing the “.in” with “.out”.

The “mytesting.txt” report should have a brief (10 word or less) description of the purpose of each test case.

6. 15% of the score will come from a physical organization that reflects a logical design in which separate problems are solved by separate modules, and internal documentation:

- Quality (clarity, accuracy, completeness, etc.) of revision history.
- Consistant indentation.
- Procedures/functions: What each does in terms of parameter register contents, return value, and action on any other data it uses.
- Header files containing interface definitions and documentation only, and other CSI333 software engineering standards in the lecture notes.
- The dependencies are accurately coded in the Makefile.
- The Makefile must be “simple” in the sense of Lab Exercise 3. DO NOT SUBMIT a “template” from a previous or other course! If you are a real Make expert, creating a simple makefile for this project will be easy. If not, follow the directions of Lab Exercise 3.