Assignment (20 points)

While studying *SPLReader.cpp* (the **"driver" program** that mushrooms-over-time into the *SPL* compiler) and *SPL.h* (*SPL* **reader-and-lister classes**) program source code—and seeing the code segments in the following bulleted-list—you were probably sorely tempted to make a few "tweaks" to the code to "make it yours". My advice? Yield to the temptation! Consider making changes to any or all of the following statements, any other statements you want to change, and even the name of the file, to make both *SPLReader.cpp* and *SPL.h* (or whatever you're gonna call 'em) "yours" (smile). **Enjoy!**

- `class SPLEXCEPTION` (in the file ***SPL.h*** **`SPLEXCEPTION`** **class**)
- `LISTER(const int LINESPERPAGE = 55);` (in the file ***SPL.h*** **`LISTER`** **class**)
- `strcat(this->sourceFileName,".spl");`
- `strcat(fullFileName,".list");`
- `if ( !LIST ) throw( SPLEXCEPTION("Unable to open list file") );`
- `LIST << setw(4) << sourceLineNumber << "  " << sourceLine << endl;`
- `LIST << FF << '"' << sourceFileName << "\" Page " << setw(4) << pageNumber << endl;`
- `LIST << "Line Source Line" << endl;`
- `LIST << "---- -----------------------------------------------------------------------------" << endl;`
- `template <int CALLBACKSALLOWED = 5>` (in the file ***SPL.h*** **`READER`** **class**)
- `READER(const int SOURCELINELENGTH = 512,const int LOOKAHEAD = 0);`
- `strcat(fullFileName,".spl")`
- `if ( !SOURCE ) throw( SPLEXCEPTION("Unable to open source file") );`
- `// Only non-printable characters allowed are EOPC,'\n', and '\t', others are changed to ' ' (and the logic that "backs-up" the comment's claim, of course)`
- `throw( SPLEXCEPTION("GetLookAheadCharacter() index out-of-range") );`
- `throw( SPLEXCEPTION("Too many callback functions") );`
- `lister->ListInformationLine("******* Source line too long!");`
- `const int SOURCELINELENGTH = 512;` (in **"driver" program file** ***SPLReader.cpp***)
- `const int LOOKAHEAD        =   2;`
- `const int LINESPERPAGE     =  60;`
- `#include "..\SPL.h"`
- `cout << "Source filename? ";`
- `cout << "SPL exception: " << splException.GetDescription() << endl;`
- `lister.ListInformationLine("******* SPL reader ending");`
- `cout << "SPL reader ending\n";`
- `cout << setw(4) << sourceLineNumber << "  ";`
- `cout << sourceLine << endl;`

Before proceeding, you <u>should</u> stop to consider whether you need to and/or want to make any other appropriate changes. For example, it makes sense to Dr. Hanna to rename *SPLReader.cpp* to *YPLReader.cpp* since this source file is the beginning of your *YPL* compiler! *YPLReader.cpp* is the basis for *YPLScanner.cpp*, which is the basis for *YPLParser.cpp*, which is the basis for *YPLCompiler.cpp*. *YPLCompiler.cpp* is the <u>ultimate goal</u> for all of your CS3335 work this semester!!!

Now, use your nascent[1] compiler to "compile" your "Hello, world" program. Note *SPLReader.cpp* uses a macro named `TRACEREADER` (see the following code segment) that controls the trace output generated by every reference to *READER::GetNextCharacter()*. When you no longer need the `TRACEREADER` "clutter" in your list file, you are invited to comment-out the macro's definition.

```
#define TRACEREADER
```

Send an email to AHanna@StMaryTX.edu with attachments that contain (1) *AnswerSheet1-to-50.txt*; (2) your version of the "driver" program file combination *SPLReader.cpp+SPL.h*; and (3) the *YPL* source file for the program you chose to "compile" to test your "driver" program

Introduction

A single-pass compiler (a language translator) reads **source code** line-by-line from a **source file** and produces both a **list file** and an **object file**. The source file is a text file that contains program components written in a specific programming language. The list file is a text file that contains a recapitulation of the source code interspersed with translation artifacts that are specific to the compiler. The object file is often a binary file that contains machine instructions and binary-coded data for a specific **computing platform** such that, when executed on this **target machine**, has that same **semantics** as (means the same thing as) the original source code.

SPL **`LISTER`** Class

The `LISTER` class outputs page-formatted information to the list file as a "log" of the translation process of a specific source file. Each page is identified at the top of the page with the source file name and a sequential page number. Source code lines (with sequential line numbers) are echoed automatically to the list file. Other useful information may also be output to the list file. The list file is named with the first name of its source file and an extension *.list*.

SPL **`READER`** Class

SPL source files, like all **free-form** programming language source files, are composed of one or more source lines contained in a text file.

> In computer programming, a **free-form** language is a programming language in which the positioning of characters on the page in the program text is insignificant. Program text does not need to be placed in specific columns as on old punched card systems, and frequently end-of-lines are insignificant. **Whitespace characters** are used only to delimit **tokens** and have no other significance. [For the **"C" locale**, whitespace characters are any of the following

`' '`	`(0x20)`	`space (SPC)`
`'\t'`	`(0x09)`	`horizontal tab (TAB)`
`'\n'`	`(0x0A)`	`newline (LF)`
`'\v'`	`(0x0B)`	`vertical tab (VT)`
`'\f'`	`(0x0C)`	`feed (FF)`
`'\r'`	`(0x0D)`	`carriage return (CR)`

Note `SPC`, `TAB`, and `LF` are the only whitespace that the *SPL* `READER` class recognizes outside of string

[1] From my dictionary, "nascent [Latin *nāscī* to be born; to arise] just beginning to exist or develop."

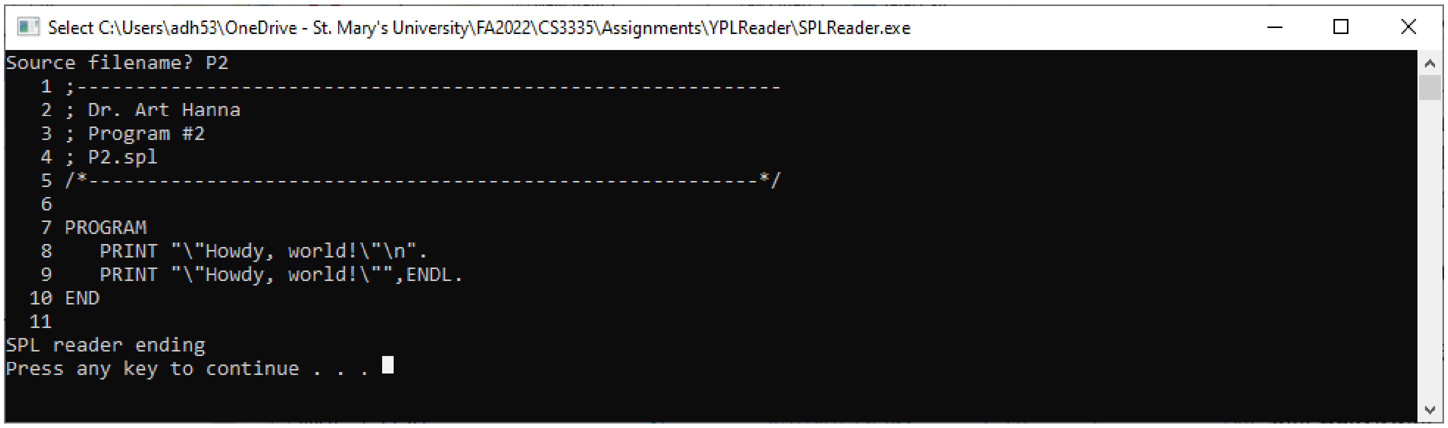
literals.]

Most free-form languages descend from ALGOL, including C/C++, Pascal, and Perl. Lisp languages are free-form, although they do not descend from ALGOL. Rexx is mostly free-form, though in some cases whitespace characters are concatenation operators. SQL, though not a full programming language, is also free-form. Python is an example of a modern programming language that is not wholly free-form.

Most free-form languages are also **structured programming languages**. Earlier imperative programming languages such as Fortran 77 used particular columns for line numbers that most structured languages don't use or need. Structured languages exist that are not free-form; for example, such as ABC, Curry, Haskell, Python and others. Many of these use some variant of the **off-side rule**, in which indentation, rather than keywords or braces, is used to group blocks of code. Based on http://en.wikipedia.org/wiki/Free-form_language on May 13, 2015.

A *SPL* source file must be named using an extension *.spl*. Every source line (except, perhaps, that last source line in the source file) is terminated with an actual **end-of-line** character. The source file is terminated with `EOPC`, the **end-of-program pseudo-character**. The `READER` class provides one-character-at-a-time, top-to-bottom, left-to-right traversal of the characters contained in the source file. The characters returned include pseudo-characters to indicate both end-of-lines (**`EOLC`**) and the end-of-program (**`EOPC`**). The `READER` class also provides multiple-character **look-ahead** (when required) to aid the **scanning** and **lexical analysis** functions.

Sample Program Dialog



Select C:\Users\adh53\OneDrive - St. Mary's University\FA2022\CS3335\Assignments\YPLReader\SPLReader.exe

```
Source filename? P2
   1 ;-------------------------------------------------------------
   2 ; Dr. Art Hanna
   3 ; Program #2
   4 ; P2.spl
   5 /*-----------------------------------------------------------*/
   6
   7 PROGRAM
   8    PRINT "\"Howdy, world!\"\n".
   9    PRINT "\"Howdy, world!\"",ENDL.
  10 END
  11
SPL reader ending
Press any key to continue . . .
```

Computational and Critical Thinking Questions (30 points)

1. Fact Two of the benefits of **object-oriented programming** are (1) the **encapsulation** of class members—both data members and member functions; and (2) **information hiding**. Using the old-fashioned **procedural programming** approach you were taught when you learned to program in C would force you to "blow-up" the `LISTER` class definition. T or F? To develop a procedurally programmed version of the `LISTER` class's functionality, the `LISTER` class data members (see the following code segment) would have to be defined as **global variables** (ugh!) and the `LISTER` class member functions would have to be defined as global functions.

```
private:
   const int LINESPERPAGE;
```

T

```
ofstream LIST;
int pageNumber;
int linesOnPage;
char sourceFileName[80+1];
```

Hint In computer science terminologies, **imperative programming** is a programming paradigm that describes computation in terms of statements that change a program state. Imperative programs define sequences of commands for the computer to perform and are, therefore, focused on describing how a program operates. (The term is often used in contrast to **declarative programming**, which focuses on what the program should accomplish without prescribing how to do it in terms of sequences of actions to be taken.)

Procedural programming is imperative programming in which the program is built from one or more **procedures** (also known as **subroutines** or **functions** [more generally, **modules**]). The terms [procedural and imperative] are often used as synonyms, but the use of procedures has a dramatic effect on how imperative programs appear and how they are constructed. Heavily-procedural programming, in which state changes are localized to procedures or restricted to explicit arguments and returns from procedures, is known as **structured programming**. From the 1960s onwards, structured programming and modular programming in general have been promoted as techniques to improve the maintainability and overall quality of imperative programs. Object-oriented programming extends this approach.

Procedural programming could be considered a step towards **declarative programming**. A programmer practicing **abstraction** can often tell, simply by looking at the names, arguments, and return types of procedures (and related comments) what a particular procedure is supposed to do, without necessarily looking at the details of how it achieves its result. At the same time, a complete program is still imperative since it 'fixes' the statements to be executed and their order of execution to a large extent. Based on http://en.wikipedia.org/wiki/Imperative_programming on May 13, 2015.

2. (Continuing 1) T or F? You would be forced to give the same answer when the T or F? Question above was asked about the `READER` class.

T

3. How many lines-per-page does *SPLReader.cpp* **`lister`** object use? Hint See the following code segment. A: 55 B: 60 C: some other number

```
LISTER(const int LINESPERPAGE = 55);

LISTER lister(LINESPERPAGE);
```

B

4. (Continuing 3) T or F? Both `LISTER` member functions *ListInformationLine()* and *ListSourceLine()* output exactly one line to the list file each time they are called.

T

5. T or F? The list file is opened by the `LISTER` constructor.

F

6. (Continuing 5) T or F? The list file is closed by the `LISTER` destructor. Hint See the following code segment.

T

```
if ( LIST.is_open() ) LIST.close();
```

7. (Continuing 6) How and why does the `LISTER` class destructor **"guard"** the *close()* function reference? Hint In computer programming, a guard is a boolean expression that must evaluate to `true` if the program execution is to continue in the branch in question. Regardless of which programming language is used, guard code or a guard clause is a check of integrity pre-conditions used to avoid errors during execution. See https://en.wikipedia.org/wiki/Guard_(computer_science) on June 27, 2017 for more detail.

it checks to make sure there is a list file open first before attempting to close it

8. How many lines are output to the list file each time the `LISTER` member function *ListTopOfPageHeader()* is executed? A: 1 B: 2 C: 3 D: 4 E: 5 or more

C

FYI #1 `LISTER` does not include the date-and-time of compilation into the top-of-page header, but it makes sense to do so. Here's how http://www.cplusplus.com/reference/ctime/asctime/. Add date-and-time to the top-of-page header if you think it's a good idea.

9. (Continuing 8) What is the significance of the output of the **`FF` control character** (see the following code segment) to the list file by the `LISTER` member function *ListTopOfPageHeader()*?

```
const char FF = 0X0C;
```

it begins the page with an empty line before outputting the top of page header

10. (Continuing 8) What is the significance of the `endl` stream manipulator reference (see the following code segment) by the `LISTER` member functions *ListTopOfPageHeader()*, *ListSourceLine()*, and *ListInformationLine()*?

```
LIST << setw(4) << sourceLineNumber << " " << sourceLine << endl;
```

it outputs a new line character to the list file and indicates the end of the line

11. The object-oriented approach to dealing with the **exceptional condition** caused by an open-list-file error "caught" by the `LISTER` class *OpenFile()* code segment is shown in the following code segment. What would happen at run-time if the *SPLReader.cpp* did not "`catch`" *SPL* exceptions?

the entire program would terminate when it encounters the error

```
LIST.open(fullFileName,ios::out);
if ( !LIST.is_open() ) throw( SPLEXCEPTION("Unable to open list file") );
```

12. How do the `LISTER` member functions *ListInformationLine()* and *ListSourceLine()* determine when to call the `LISTER` member function *ListTopOfPageHeader()*?

it counts the number of lines on the page and if that number is greater than the maximum number of lines per page specified by the variable LINESPERPAGE, then it will begin a new page and call listTopOfPageHeader();

13. The `LISTER` member function *ListTopOfPageHeader()* is a
A: constructor B: destructor C: accessor D: mutator E: utility

E

FYI #2 You should be able to explain all the ways in which the `READER` class constants `EOLC` and `SOURCELINELENGTH` are different from each other.

```
public:
   static const char EOLC = '\n';

private:
   const int SOURCELINELENGTH;
```

14. Why is the `NEXTCHARACTER` class defined with `struct` instead of `class`?

because with a struct the attributes are automatically public
and there is no need to include private member functions

15. What does this syntax mean? Hint Does this make the `READER` class a **generic**, that is, a **class template**?

```
template <int CALLBACKSALLOWED = 5>
```

it means the reader class is a generic class template and if no value for CALLBACKSALLOWED is specified, then it is defaulted to 5

16. Express the value of `READER::EOPC` as a 2-nibble hexadecimal integer______.

```
static const char EOPC = 0;
```

00

FYI #3 Question What are **call-back functions** and why and when and by "whom" are they called?

```
    int numberCallbacks;
    void (*CallbackFunctions[CALLBACKSALLOWED+1])
       (int sourceLineNumber,const char sourceLine[]);
```

How does your "whom" answer setup call-back functions? Hint

```
    void AddCallbackFunction(void (*CallbackFunction)
       (int sourceLineNumber,const char sourceLine[]));
```

17. What is the maximum-length-of-a-source-line used by the *SPLReader.cpp* `reader` object______? Hint

```
const int SOURCELINELENGTH        = 512;

READER(const int SOURCELINELENGTH = 512,const int LOOKAHEAD = 0);

READER reader(SOURCELINELENGTH,LOOKAHEAD);
```

514

18. (Continuing 17) What is the number of look-ahead characters used by the `reader` object______? Hint

```
const int LOOKAHEAD               =   2;

READER<CALLBACKSUSED> reader(SOURCELINELENGTH,LOOKAHEAD);

READER(const int SOURCELINELENGTH = 512,const int LOOKAHEAD = 0);
```

2

19. (Continuing 18) What is the size of the `READER` class data member `nextCharacters` (a dynamically-allocated array) when `LOOKAHEAD` is 0? ______ Hint When `LOOKAHEAD` is 0, are there any characters in the `nextCharacters` array to look-ahead at?!

FYI #4 You should be able to explain how character look-ahead "works".

1

20. (Continuing 19) T or F? When `(LOOKAHEAD > 0)` then it is possible to have more than one `EOPC` character in the `nextCharacters[]` array both at the start of program execution (before any source lines are input) and at the end of program execution. Hint See *READER::OpenFile()*.

T

FYI #5 (Fact #1) Every source file "ends" with an end-of-program **pseudo-character**, **`EOPC`**. (Fact #2) Every source code line contained in a source file ends with an end-of-line character, **`EOLC`**, except, perhaps, the last line in source file (for example, `'\n'` is the actual stored end-of-line character used by Windows); that is, the last line, `xxx`, of a source file may end with or without an end-of-line character, namely, `xxx EOLC EOPC` or `xxx EOPC`. Explain how these two facts complicate the logic of the `READER` class member functions *GetNextCharacter()* and *ReadSourceLine()*.

21. T or F? The `READER` class object `reader` **has-a (contains-a)** `LISTER` data member accessed indirectly using the `LISTER` pointer `lister`. Hint

F

```
LISTER *lister;
void SetLister(LISTER *lister);
```

FYI #6 Explain why the `READER` class data member `lister` must be a pointer and not a reference or a bona fide copy of the global object `lister`.

22. Explain what the `READER` class *GetNextCharacter()* member function does in the following code segment.

```
for (int i = 1; i <= LOOKAHEAD; i++)
   nextCharacters[i-1] = nextCharacters[i];
```

it "drops off" previously read characters and replaces them with the next characters to be read

23. T or F? The `READER` class *GetNextCharacter()* member function code segment shown below must refer to `READER EOPC` constant using the **scope resolution operator** `::`.

```
if ( atEOP )
   character = READER::EOPC;
```

F

24. (Continuing 18) Which index is used to place each source character when it is first stored into the `nextCharacters[]` array during *SPLReader.cpp* execution? A: 0 B: 1 C: 2 D: 3 E: none of these

A

25. T or F? There is an `EOLC` stored in the character array `sourceLine[]` for every source line read from the source file by the `READER` class *ReadSourceLine()* member function. F

26. (Continuing 25) The *ReadSourceLine()* member function of the `READER` class attempts to read a source line from the source file into the character array `sourceLine[]` whose size is fixed when the `READER` object is constructed. What size does *SPLReader.cpp* use______? 512

27. (Continuing 26) What happens in *ReadSourceLine()* when a source line is too big to "fit" into the `sourceLine[]` array? Hint Make `SOURCELINELENGTH` small enough to see what happens!

it fits as much of the source line as it can into the array and then outputs the rest on the next line

28. Explain how the *ReadSourceLine()* loop shown in the following code segment "works".

```
// Erase *ALL* control characters at end of source line (if any)
   while ( (0 <= (int) strlen(sourceLine)-1) &&
           iscntrl(sourceLine[(int) strlen(sourceLine)-1]) )
      sourceLine[(int) strlen(sourceLine)-1] = '\0';
```

it checks if the length of the source line is greater than 0, and if the next to last character is a control character. If both are true, it changes the next to last character to a null character

FYI #7 The **macro `TRACEREADER`** is used to conditionally-compile some instrumented output to the list file. Describe the `TRACEREADER`-enabled output. How do you "turn-off" the trace output?

29. T or F? *SPLReader.cpp* must refer to `READER EOPC` constant exactly as shown in the following code segment including the highlighted template parameter in pointy brackets and the scope resolution operator `::`.

Hint Compare with Question 23. T

```
do
{
   nextCharacter = reader.GetNextCharacter();
} while ( nextCharacter.character != READER<CALLBACKSUSED>::EOPC );
```

30. T or F? The relative order of the two *SPLReader.cpp main()* function local data definitions shown in the following code segment can be changed with impunity. T

```
READER<CALLBACKSUSED> reader(SOURCELINELENGTH,LOOKAHEAD);
LISTER lister(LINESPERPAGE);
```

FYI #8 Why is the order of the first three statements shown in the following code segment taken from the *SPLReader.cpp main()* function important to retain?

```
lister.OpenFile(sourceFileName);
reader.SetLister(&lister);
reader.AddCallbackFunction(Callback1);
reader.AddCallbackFunction(Callback2);
reader.OpenFile(sourceFileName);
```

Compare *SPLReader.cpp* with the C-style, old-school version (*SPLReader.c*) from which it is derived to answer the following *FYI* questions.

FYI #9 What **globally-defined variables** does *SPLReader.cpp* use? *SPLReader.c*? Why is there such a significant difference? Hint Can you say **"encapsulation"**?

FYI #10 Which version of the "reader" program is shorter? Why? Which version do you prefer? Why?

FYI #11 How can you convert the **macro definitions** in *SPLReader.c* (see the following code segment) into **globally-defined constants** like those scattered around *SPLReader.cpp*? What advantages, if any, are there to converting a macro definition to a global constant?

```
#define SOURCELINELENGTH   512
#define LINESPERPAGE        55
#define LOOKAHEAD            3
#define EOPC                 0
#define EOLC              '\n'
#define TABC              '\t'
```

FYI #12 Complete the sentence, "In general, the initialization of globally-defined variables at the *SPLReader.c* start-of-program is accomplished in *SPLReader.cpp* by…"

FYI #13 Where is the termination (end-of-program) processing of *SPLReader.c* (see the following code segment) accomplished in *SPLReader.cpp*?

```
fclose(SOURCE);
fclose(LIST);
```

FYI #14 Which approach do you prefer, *SPLReader.c* (old school) or *SPLReader.cpp*? Why?

```
;-------------------------------------------------------------
; Dr. Art Hanna
; Program #2
; P2.spl
/*-----------------------------------------------------------*/

PROGRAM
   PRINT "\"Howdy, world!\"\n".
   PRINT "\"Howdy, world!\"",ENDL.
END

//-------------------------------------------------------------
// Dr. Art Hanna
// SPL Reader, "driver" program
// SPLReader.cpp
//-------------------------------------------------------------
#include <iostream>
#include <iomanip>

#include <fstream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cctype>
#include <vector>

using namespace std;

#define TRACEREADER

#include "..\SPL.h"

//-------------------------------------------------------------
int main()
//-------------------------------------------------------------
{
   void Callback1(int sourceLineNumber,const char sourceLine[]);
   void Callback2(int sourceLineNumber,const char sourceLine[]);

   char sourceFileName[80+1];
   NEXTCHARACTER nextCharacter;

   READER<CALLBACKSUSED> reader(SOURCELINELENGTH,LOOKAHEAD);
   LISTER lister(LINESPERPAGE);

   cout << "Source filename? ";
   cin >> sourceFileName;

   try
   {
      lister.OpenFile(sourceFileName);
      reader.SetLister(&lister);
      reader.AddCallbackFunction(Callback1);
      reader.AddCallbackFunction(Callback2);
      reader.OpenFile(sourceFileName);

      do
      {
         nextCharacter = reader.GetNextCharacter();
      } while ( nextCharacter.character != READER<CALLBACKSUSED>::EOPC );
   }
   catch (SPLEXCEPTION splException)
   {
      cout << "SPL exception: " << splException.GetDescription() << endl;
   }
   lister.ListInformationLine("******* SPL reader ending");
   cout << "SPL reader ending\n";

   system("PAUSE");
   return( 0 );
}

//-------------------------------------------------------------
void Callback1(int sourceLineNumber,const char sourceLine[])
//-------------------------------------------------------------
{
   cout << setw(4) << sourceLineNumber << " ";
}

//-------------------------------------------------------------
void Callback2(int sourceLineNumber,const char sourceLine[])
//-------------------------------------------------------------
{
   cout << sourceLine << endl;
}
```