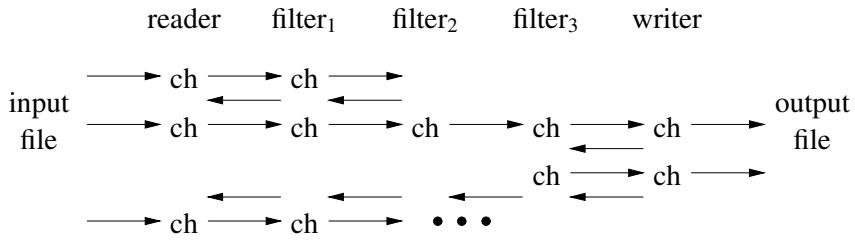
CS 343 Winter 2023 – Assignment 2
Instructor: Caroline Kierstead
Due Date: Monday, February 6, 2023 at 22:00
Late Date: Wednesday, February 8, 2023 at 22:00

January 26, 2023

This assignment examines complex semi-coroutines, and introduces full-coroutines and concurrency in μC++. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution. Unless otherwise specified, writing a C-style solution for questions is unacceptable and receives little or no marks. (You may freely use the code from these example programs.)

1. Write a program that filters a stream of text. The filter semantics are specified by command-line options. The program creates a *semi-coroutine* filter for each command-line option joined together in a pipeline with a reader filter at the input end of the pipeline, followed by the command-line filters in the middle of the pipeline (maybe zero), and a writer filter at the output end of the pipeline. Control passes from the reader, through the filters, and to the writer, ultimately returning back to the reader. One character moves along the pipeline at any time. For example, a character starts from the reader filter, may be deleted or transformed by the command-line filters, and any character reaching the writer filter is printed.



(In the following, you may not add, change or remove prototypes or given members; you may add a destructor and/or private and protected members.)

Each filter must inherit from the abstract class Filter:

```
_Coroutine Filter {
  protected:
    _Event Eof {};                      // no more characters
    Filter * next;                      // next filter in chain
    unsigned char ch;                   // communication variable
  public:
    Filter( Filter * next ) : next( next ) {}
    void put( unsigned char c ) {
        ch = c;
        resume();
    }
};
```

which ensures each filter has a put routine that can be called to transfer a character along the pipeline.

The reader reads characters from either standard input or the input file specified on the command-line. It passes these characters to the first coroutine in the filter:

```
_Coroutine Reader : public Filter {
    // YOU MAY ADD PRIVATE MEMBERS
    void main();
  public:
    Reader( Filter * f, istream & i );
};
```

The reader constructor is passed the next filter object, which the reader passes one character at a time from the input stream, and an input stream object from which the reader reads characters. No coroutine calls the put routine of the reader; all other coroutines have their put routine called. When the reader reaches end-of-file, it raises the exception Eof at the next filter, resumes the next filter with an arbitrary character, and terminates.

The writer is passed characters from the last coroutine in the filter pipeline and writes these characters to either standard output or the output file specified on the command-line:

```
_Coroutine Writer : public Filter {
    // YOU MAY ADD PRIVATE MEMBERS
    void main();
  public:
    Writer( ostream & o );
};
```

The writer constructor is passed an output stream object to which this filter writes characters that have been filtered along the pipeline. No filter is passed to the writer because it is at the end of the pipeline. When the write receives the Eof exception, it prints out how many characters where printed, e.g.:

```
16 characters
```

All other filters have the following interface:

```
_Coroutine filter-name : public Filter {
    // YOU MAY ADD PRIVATE MEMBERS
    void main();
  public:
    filter-name( Filter * f, ... );
};
```

Each filter constructor is passed the next filter object, which this filter passes one character at a time after performing its filtering action, and "..." is any additional information needed to perform the filtering action. When a filter reaches end-of-file, it raises the exception Eof at the next filter, resumes the next filter with an arbitrary character, and terminates.

The pipeline is built by the program main from writer to reader, in reverse order to the data flow. Each newly created coroutine is passed to the constructor of its predecessor coroutine in the pipeline. The reader's constructor resumes itself to begin the flow of data, and it calls the put routine of the next filter to begin moving characters through the pipeline to the writer. Normal characters, as well as control characters (e.g., `'\n'`, `'\t'`), are passed through the pipeline. When the reader reaches end-of-file, it raises the exception Eof at the next filter, resumes the next filter with an arbitrary character, and terminates. Similarly, each coroutine along the filter pipeline raises the exception Eof at the next filter along the pipeline and then terminates. The program main ends when the reader declaration completes, implying all the input characters have been read and all the filter coroutines are terminated. The reader coroutine can read characters one at a time or in groups; the writer coroutine can write characters one at a time or in groups.

Filter options are passed to the program via command line arguments. For each filter option, create the appropriate coroutine and connect it into the pipeline. If no filter options are specified, then the output should simply be an echo of the input from the reader to the writer. ***Assume all filter options are correctly specified, i.e., no error checking is required on the filter options.***

The filter options that must be supported are:

–c [l | u] The *case* option changes the case of letters to either lower for an argument of "l" or upper case for an argument of "u" (see man isupper and man tolower). (Assume whitespace separates –c and l or u.)

-w The *whitespace* option removes all spaces and tabs (isblank) from the start and end of lines, collapses multiple spaces and tabs within a line into a single space, and eliminates empty lines (consisting only of whitespace). Lines are delimited by the newline character (`'\n'`).

-H *style wordlist #* The highlight option emphasizes each occurrence in the input stream of every word in the *wordlist*. If the *style* is *bold* then words in *wordlist* occurring in the input stream are made to appear in bold face when displayed on a terminal. Similarly, if the *style* is *underline* then each occurrence is underlined. Only the first word after -H is interpreted as a style. Each word in the *wordlist* is separated from the next by whitespace, including the number sign. The *wordlist* is terminated by the number sign, even if it is empty. For simplicity, only upper and lower case letters occur in the *wordlist*. For similar reasons, do not worry about correctly handling the case where the word list from separate highlight options contain overlaps. An example of the command-line follows:

```
./filter -H bold cat dog '#'
```

For vt100 compatible terminals (like default xterms under X-Windows) highlights can be started and stopped with special key sequences placed in the character stream. The starting sequences for *bold* and *underline* are \E[1m and \E[4m respectively. The ending sequence is \E[m for both. The character pair \E in the sequences stands for the single character escape key ESC ('\x1b'). Use these sequences for highlighting.

The filter only considers words to be sequences of alphabetic characters, and does not look for the word to be a substring in any other sequence on non-whitespace characters. Punctuation and whitespace delimit words. For example, if the filter was looking for the word "dog", input such as

```
1234dog1234 abcdogabc dogmatic
  1dog    dog1
dog1dog{
```

would not be highlighted, but

```
((dog)) dog .dog
___dog___
```

would be.

The order in which filter options appear on the command line is significant, i.e., the left-to-right order of the filter options on the command line is the first-to-last order of the filter coroutines. As well, a filter option may appear more than once in a command. Each filter should be constructed without regard to what any other filters do, i.e., there is no communication among filters using global variables; all information is passed using member put. **Hint:** scan the command line left-to-right to locate and remember the position of each option, and then scan the option-position information right-to-left (reverse order) to create the filters with their specified arguments.

The executable program is to be named filter and has the following shell interface:

```
filter [ -filter-options ... ] [ infile [outfile] ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) ***If filter options appear, assume they appear* before *the file names.*** Terminate the program for unknown or insufficient command arguments, or if an input or output file cannot be opened. Assume any given argument values are correctly formed, i.e., no error checking is required for numeric values. If no input file name is specified, input from standard input and output to standard output. If no output file name is specified, output to standard output.

Hint: You may find the sample program, https://student.cs.uwaterloo.ca/~cs343/examples/uIO.cc, useful.

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. A state variable contains information that is not part of the computation and exclusively used for control-flow purposes (like flag variables). Use of execution state variables in a coroutine usually indicates you are not using the ability of the coroutine to remember prior execution information. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 3.1.3 in the Course Notes for details on this issue. Also, make sure a coroutine's public methods are used for passing information to the coroutine, but not for doing the coroutine's work, which must be done in the coroutine's main.

2. Write a *full coroutine* to play the card game "Schmilblick". Schmilblick is played with a variable-sized deck of cards, and each player takes a number of cards from the deck and pass the remaining deck to the player on the left if the number of remaining cards is odd, or to the right if the number of remaining cards is even. A player must take at least one card and no more than a certain maximum; a player cannot take more cards than are in the deck. *After making a play*, a player checks to see if they received a deck that is a multiple of 7 cards, called a "death deck"; if so, the player removes themself from the game and no longer participates. Otherwise, a player always makes a play, except when they are the last player. (If a play is not made, the death deck would be passed to all players.) The player who takes the last cards or is the only player remaining wins the game, unless the last player receives the death deck and then no one wins.

   At random times, 1 in 10 plays, a player yells "Schmilblick" and every player must take a drink. A round of Schmilblick is implemented by the Schmilblick raiser having a drink and then raising the Schmilblick *resumption* exception at the player on the right, and so on until all players receive a Schmilblick exception. After all the players have had a drink, the game continues exactly where it left off. Note, a player that received a death deck cannot start or participate in a Schmilblick!

   The interface for a Player is (you may only add a public destructor and private members):

```
_Coroutine Player {
    // YOU MAY ADD PRIVATE MEMBERS, INCLUDING STATICS
  public:
    enum { DEATH_DECK_DIVISOR = 7 };
    static void players( unsigned int num );
    Player( PRNG & prng, unsigned int id, Printer & printer );
    void start( Player &lp, Player &rp );
    void play( unsigned int deck );
    void drink();
};
```

   The players routine is called before any players are created to set the total number of players in the game. Then players are created, where the constructor is passed a reference to a printer object and an identification number assigned by the main program. (Use values in the range 0 to $N-1$ for identification values.) To form the circle of players, the start routine is called for each player from the program main to pass a reference to the player on the left and right. The start member also resumes the player to set the program main as its starter so the last player can get back to the program main at the end the game. The play routine receives the deck of cards passed among the players. The drink routine resumes the coroutine to receive the non-local exception.

   All game output must be generated by calls to a printer class. Two blank lines are printed between games and may be printed by the printer or main driver loop, but there must NOT be blank lines following the last game. The interface for the printer is (you may add only a public destructor and private members):

```
class Printer {
    // YOU MAY ADD PRIVATE MEMBERS
  public:
    Printer( const unsigned int NoOfPlayers, const unsigned int NoOfCards );
    void prt( unsigned int id, int took, unsigned int RemainingPlayers );  // card play
    void prt( unsigned int id );  // drink (Schmilblick)
};
```

   The printer attempts to reduce output by condensing the play along a line. Figure 1 shows example outputs from different runs of the program. Each column is assigned to a player, and a column entry indicates a player is making a play (C:Rd) or having a drink (D). Making a play is a triple of values:

   (a) the number of (C)ards taken by a player,
   (b) the number of cards (R)emaining in the deck,
   (c) the (d)irection the remaining deck is passed, where "<" means to the left, ">" means to the right, "X" means the player terminated, and "**#**" means the game is over. If a game ends because a player takes the last cards, the output has a single "**#**" appended. If a game ends because there are no more players, the output is "**#***deck-size***#**", where "*deck-size*" is the number of cards last passed to this player. It is possible for the last player to simultaneously receive a death deck, and hence lose the game, indicated by game end and death, e.g., 7:0#X or #14#X.

Last Card Taken

Players: 3		Cards: 46
P0	P1	P2
6:26>	8:38>	6:32>
6:15<X	5:21<	7:8>
	2:6>	6:0#

No Players Remaining

Players: 3		Cards: 87
P0	P1	P2
1:84>	2:85<	
3:70>	6:78>X	5:73<
#67#		3:67<X

Last Card / Death

Players: 3		Cards: 59
P0	P1	P2
5:44>X	6:49<	4:55<
	5:39<	6:33<
	8:25<	6:19<
	4:15<	8:7<
	7:0#X	

Drinks

Players: 3		Cards: 33
P0	P1	P2
4:27<	2:31<	7:20>
D	D	D
1:19<	3:10>	6:13<
2:4>	4:0#	4:6>

Figure 1: Schmilblick Card-Game Output

Player information is buffered in the printer until a play would overwrite a buffer value. At that point, the buffer is flushed (written out) displaying a line of player information. The buffer is cleared by a flush so previously stored values do not appear when the next player flushes the buffer and an empty column is printed. All column spacing can be accomplished using the standard 8-space tabbing, i.e., do NOT use spaces to align columns. **Store information in internal format for flushing; do NOT build and store C/C++ strings of text for output**.

Some students find the printer difficult to implement, so begin by having the printer just print one line for each print call. **After** the card game is working, come back to the printer and start working on buffering the output. You may hand-in this program, even if it does not match the given sample output, and receive most of the marks.

The main program plays games games sequentially, i.e., one game after the other, where games is a command line parameter. For each game, N players are created, a deck containing M cards is created, and a random player is chosen and passed the deck of cards. The player passed the deck of cards begins the game, and each player follows the simple strategy of taking C cards, where C is a random integer in the range from 1 to 8 inclusive. **Do not spend time developing strategies to win.** At the end of each game, it is unnecessary for a player's coroutine-main to terminate but ensure each player is deleted before starting the next game.

The executable program is named cardgame and has the following shell interface:

```
cardgame [ games | "d" [ players | "d" [ cards | "d" [ seed | "d" ] ] ] ]
```

games is the number of card games to be played (≥ 0). If no value for games is specified or d, assume 5.

players is the number of players in the game (≥ 2). If no value for players is specified or d, generate a random integer in the range from 2 to 10 inclusive for each game.

cards is the number of cards in the game (> 0). If no value for cards is specified or d, generate a random integer in the range from 10 to 200 inclusive for each game.

seed is the starting seed for the random number generator to allow reproducible results (> 0). If no value for seed is specified or d, initialize the random number generator with an arbitrary seed value (e.g., getpid() or time).

Check all command arguments for correct form (integers) and range; print an appropriate usage message and terminate the program if a value is missing or invalid.

To obtain repeatable results, all random numbers are generated using the μC++ class PRNG (see Appendix C in the μC++ reference manual). Create two random number generates: one for the program main and one shared by all the players; both random generators are initialized to the same seed. The program-main generator has up to three calls depending on the command-line arguments: the number of players for a game, the number of cards in the initial deck of cards, and the random player to start the game (in that order). Each player makes two calls to the player generator to make a random play and start a Schmilblick (in that order). All random rolls (1 in N chance) are generated using prng(N) == 0.

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. A state variable contains information that is not part of the computation and exclusively used for control-flow purposes (like flag variables). Use of execution state variables in a coroutine usually indicates you are not using the ability of the coroutine to remember prior execution information. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 3.1.3 in the Course Notes for details on this issue. Also, make sure a coroutine's public methods are used for passing information to the coroutine, but not for doing the coroutine's work, which must be done in the coroutine's main.

```
#include <iostream>
using namespace std;

static volatile long int shared = -6;          // volatile to prevent dead-code removal
static intmax_t iterations = 500000000;

_Task increment {
    void main() {
        for ( int i = 1; i <= iterations; i += 1 ) {
            shared += 1;                       // two increments to increase pipeline size
            shared += 2;
        } // for
    } // increment::main
}; // increment

static intmax_t convert( const char * str ); // copy from https://student.cs.uwaterloo.ca/~cs343/examples/uIO.cc

int main( int argc, char * argv[] ) {
    intmax_t processors = 1;
    try {                                      // process command-line arguments
        switch ( argc ) {
          case 3: processors = convert( argv[2] ); if ( processors <= 0 ) throw 1;
          case 2: iterations = convert( argv[1] ); if ( iterations <= 0 ) throw 1;
          case 1: break;                       // use defaults
          default: throw 1;
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ iterations (> 0) [ processors (> 0) ] ]" << endl;
        exit( 1 );
    } // try
    uProcessor p[processors - 1];              // create additional kernel threads
    {
        increment t[2];
    } // wait for tasks to finish
    cout << "shared:" << shared << endl;
} // main
```

Figure 2: Interference

3. Compile the program in Figure 2 using the u++ command with compilation flag -multi, and 1 and 2 processors.

   (a) Perform the following experiments.

   - Run the program 10 times with command line argument 500000000 1 on a multi-core computer with at least 2 CPUs (cores).
   - Show the 10 results.
   - Run the program 10 times with command line argument 500000000 2 on a multi-core computer with at least 2 CPUs (cores).
   - Show the 10 results.

   (b) Must all 10 runs for each version produce the same result? Explain your answer.

   (c) In theory, what are the smallest and largest values that could be printed out by this program with an argument of 100000000? Explain your answers. (**Hint:** one of the obvious answers is wrong.)

   (d) Explain any subtle difference between the size of the values for 1 processor and 2 processors.

Submission Guidelines

Follow these guidelines carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text or test-document file, e.g., *.{txt,testdoc} file, must be ASCII text and not exceed 500 lines in length, using the command fold -w120 *.testdoc | wc -l.** Programs should be divided into separate compilation

units, i.e., *.{h,cc,C,cpp} files, where applicable. Use the submit command to electronically copy the following files to the course account.

1. q1*.{h,cc,C,cpp} – code for question 1, p. 1. **Program documentation must be present in your submitted code. No test documentation is submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**
2. q2*.{h,cc,C,cpp} – code for question 2, p. 4. **Program documentation must be present in your submitted code. No test documentation is submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**
3. q3.txt – contains the information required by question 3.
4. Modify the following Makefile to compile the programs for question 1, p. 1 and 2, p. 4 by inserting the object-file names matching your source-file names.

```
CXX = u++                                   # compiler
CXXFLAGS = -g -Wall -Wextra -MMD -O2
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS1 = # object files forming 1st executable with prefix "q1"
EXEC1 = filter

OBJECTS2 = # object files forming 2nd executable with prefix "q2"
EXEC2 = cardgame

OBJECTS = ${OBJECTS1} ${OBJECTS2}           # all object files
DEPENDS = ${OBJECTS:.o=.d}                  # substitute ".o" with ".d"
EXECS = ${EXEC1} ${EXEC2}                   # all executables

.PHONY : all clean
.ONESHELL :

all : ${EXECS}                              # build all executables

#################################################################

${EXEC1} : ${OBJECTS1}
    ${CXX} $^ -o $@


${EXEC2} : ${OBJECTS2}                      # link step 2nd executable
    ${CXX} ${CXXFLAGS} $^ -o $@

#################################################################

${OBJECTS} : ${MAKEFILE_NAME}               # OPTIONAL : changes to this file => recompile

-include ${DEPENDS}                         # include *.d files containing program dependences

clean :                                     # remove files that can be regenerated
    rm -f *.d *.o ${EXECS}
```

This makefile is used as follows:

```
$ make filter
$ ./filter
$ make cardgame
$ ./cardgame ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make filter or make cardgame in the directory to compile the programs. This Makefile must be submitted with the assignment to build the program, so it must be correct. Use the web tool Request Test Compilation to

ensure you have submitted the appropriate files, your makefile is correct, and your code compiles in the testing environment.

Follow these guidelines. Your grade depends on it!