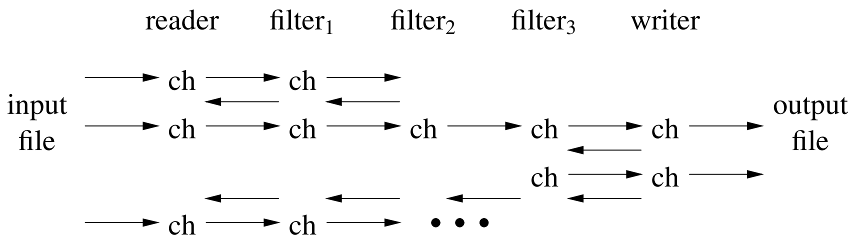
CS 343 Fall 2021 – Assignment 2
Instructor: Peter Buhr
Due Date: Wednesday, October 6, 2021 at 22:00
Late Date: Friday, October 8, 2021 at 22:00

September 17, 2021

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 will receive 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 the specified input file and 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 the specified output file:

```
_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.)

–e *number* The encrypt option uses the *Progressive Caesar cipher* encoding to rotate each letter in the input stream by *number* positions in the alphabet. If *number* is zero or positive, a letter in rotated forward by *number* modulo 26 (0..25) in lexigraphical order; otherwise it is rotated backwards by *number* modulo 26 (0..-25) in lexigraphical order. After any 13 consecutive characters, *number* is progressed by incrementing by 1 if zero or positive, or decrementing by -1 if negative. The case of each letter, either upper or lower, is retained in the encoded text, so rotations occur within the same case. Characters that are not letters pass through the filter unchanged. The encoding can be reversed by using the negation of the number used in the original encryption. Assume –e and *number* are separated by spaces.

–s The *capitalize* option capitalizes the first letter of every sentence. A sentence starts with the letter following whitespace (space, tab, newline) (isspace) characters after a period, question mark, or exclamation point. If the starting letter is lower case (islower), the filter transforms the letter to upper case. The first letter character in the pipeline is a special case and should be capitalized even though there is no preceding whitespace.

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.

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* that simulates the game of Hot Potato. What makes the potato *hot* is that it explodes after its internal count-down timer reaches zero. The game consists of *N* players linked in a circle, where one of the players also acts as the umpire. The umpire (or special case in the program main for the first toss) starts a *game* by randomly tossing the *hot* potato to a player on their left or right. The potato is then randomly tossed left or right among the players until the timer goes off and it explodes. The player holding the potato when the timer goes off (potato explodes) is removed from the game (circle) and deleted. A player cannot delete itself, so the umpire must perform this action. The last remaining player is the winner.

   Should the potato explode for the umpire, a new umpire is *elected* among the remaining players. An election involves traversing the circle using exceptions to find the remaining player with the highest id; this player becomes the new umpire.

   The potato contains a count-down timer that goes off after a random number of clock ticks. The interface for the Potato is (you may only add a public destructor and private members):

```
_Coroutine Player {
    _Event Terminate {
      public:
        Player & victim;                       // delete player
        Terminate( Player & victim ) : victim( victim ) {}
    };
    _Event Election {
      public:
        Player * player;                       // highest player id seen so far
        Election( Player * player ) : player( player ) {}
    };
    Player * partner[2];                       // left and right player
    // YOU ADD MEMBERS HERE
    void main();
    void vote();                               // resume partner to vote
    void terminate();                          // resume umpire
  public:
    static Player * umpire;                    // current umpire

    Player( PRNG & prng, int id, Potato & potato );
    void start( Player & lp, Player & rp );    // supply partners
    int getId();                               // player id
    void toss();                               // tossed potato
};
```

Figure 1: Player Interface

```
class Potato {
    // YOU ADD MEMBERS HERE
  public:
    _Event Explode {};
    Potato( PRNG & prng, unsigned int maxTicks = 10 );
    void reset( unsigned int maxTicks = 10 );
    void countdown();
};
```

The constructor is passed a PRNG and optionally the *maximum* number of ticks until the timer goes off. The potato chooses a random value between 1 and the maximum for the number of ticks, inclusive. Member reset is called by the umpire to reinitialize the timer for the next set. The constructor resets the potato. Member countdown is called by the players, and throws exception Explode, if the timer has reached zero. Rather than absolute time to implement the potato's timer, each call to countdown is one tick of the clock (relative time).

Figure 1 shows the interface for a Player (you may only add a public destructor and private members). The exception Terminate is raised at the umpire and contains the player to be deleted. The exception Election is raised at the player on the right to run an election and contains the highest player id seen so far in the circle.

The array partner contains the left/right partners of a player. **You may *not* change the array into a C++ vector.** The vote member is called to vote in an election (discussed below). The terminate member is only called for the umpire after a player receives the Explode exception from the potato. The terminating player first raises the Terminate exception at the umpire, and calls the umpire's terminate member to cause propagation of the nonlocal exception; the call to terminate never returns. When the umpire handles the Terminate exception, it unlinks the player given in the exception from the circle, deletes it, resets the potato, and continues the game by randomly tossing the potato to its left/right player.

The **static** variable umpire is the player currently acting as the umpire. This variable allows a player (and program main) to communicate with the umpire. The variable umpire is updated after electing a new umpire.

The constructor is passed a PRNG, an identification number (0 to $N-1$), and the potato created by the main program. The start member receives a player's left and right partners from the program main. The getId member returns a player's id. The toss member is called to conceptually pass the potato to another player.

When a terminated player is also the umpire, it raises an Election exception at the player on the right, and calls that player's vote member to cause propagation of the nonlocal exception. Each resumed player handles the

```
$ hotpotato 1 6 1005
6 players in the game
  POTATO goes off after 2 ticks
U 0 -> 5 is eliminated
  POTATO goes off after 1 tick
U 0 is eliminated
E 0 -> 1 -> 2 -> 3 -> 4 : umpire 4
  POTATO goes off after 2 ticks
U 4 -> 3 is eliminated
  POTATO goes off after 3 ticks
U 4 -> 1 -> 4 is eliminated
E 4 -> 1 -> 2 : umpire 2
  POTATO goes off after 1 tick
U 2 is eliminated
E 2 -> 1 : umpire 1
  POTATO goes off after 4 ticks
U 1 wins the Match!
```

Figure 2: Sample Output

Election exception, updating the election id in the exception and raising it at the player on the right. After the election finishes and a new umpire is set, control returns to the old umpire. The old umpire raises the Terminate exception at the new umpire, to tell the new umpire to terminate it, and calls the new umpire's terminate member to cause propagation of the nonlocal exception. The new umpire handles the Terminate exception as defined above. This toss counts with respect to the timer in the potato. Note, good software engineering encapsulates calls to resume only in interface members, rather than directly calling resume for another player.

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

```
hotpotato [ games | 'd' [ players | 'd' [ seed | 'd' ] ] ]
```

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

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

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

Check all command arguments for correct form (integers or d) 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 class PRNG. There are up to four calls to get random numbers: up to two in program main (only one if a value for players is specified on the command line), one in potato::reset, and one in Player::main.

The program main creates three PRNGs, seeding them all with the same seed value. The program main uses the first PRNG, passes second to potato, and passes the third to each player, so all players use the same PRNG. Then the potato and players are created in increasing order of player id, but the umpire *deletes* the players during the game. After creating the players, generate a random player index, between 0 and players – 1, and swap that position with position 0; hence players may not be in increasing order by player id. To form the ring of players, the program main calls the start member for each player to link the players together into a circle. The start member also resumes the player to set the program main as its starter (needed during termination). The main member of each player suspends back immediately so the next player can be started. Finally, the program main sets the global umpire variable to the player with id 0 (located at the random position), and tosses the potato to it to start the game. Figure 2 shows the output of a game, where U means umpire and E means election.

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*

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

static volatile long int shared = 0;        // volatile to prevent dead−code removal
static long int iterations = 100000000;

_Task increment {
    void main() {
        for ( int i = 0; i < iterations; i += 1 ) {
            shared += 1;                    // two increments to increase pipeline size
            shared += 1;
        }
    }
};
int main( int argc, char * argv[] ) {
    int processors = 1;
    try {                                   // process command−line arguments
        switch ( argc ) {
          case 3: processors = stoi( argv[2] ); if ( processors <= 0 ) throw 1;
          case 2: iterations = stoi( 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;
}
```

Figure 3: Interference

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.

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

   (a) Perforamnce the following experiments.

   - Run the program 10 times with command line argument 300000000 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 300000000 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. Output for this question is checked via a marking program, so it must match exactly with the given program.**
2. q1*.testdoc – test documentation for question 1, p. 1, which includes the input and output of your tests. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**
3. PRNG.h – random number generator (provided)
4. q2*.{h,cc,C,cpp} – code for question 2, p. 3. **Program documentation must be present in your submitted code. No user, system or 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.**
5. q3*.txt – contains the information required by question 3.
6. Modify the following Makefile to compile the programs for question 1, p. 1 and 2, p. 3 by inserting the object-file names matching your source-file names.

```
CXX = u++                                        # compiler
CXXFLAGS = -g -Wall -Wextra -MMD
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 = hotpotato

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

.PHONY : all clean

all : ${EXECS}                                   # build all executables

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

${EXEC1} : ${OBJECTS1}                           # link step 1st executable
	${CXX} ${CXXFLAGS} $^ -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 hotpotato
$ ./hotpotato ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make filter or make hotpotato 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!