CS 343 Winter 2024 – Assignment 2
Instructor: Colby Parsons
Due Date: Monday, February 5, 2024 at 22:00
Late Date: Wednesday, February 7, 2024 at 22:00

January 18, 2024

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 *semi-coroutine* to sort a set of values, which may contain duplicate values, into ascending order using a binary-tree insertion method. This method constructs a binary tree of the data values, which can subsequently be traversed to retrieve the values in sorted order. Construct a binary tree without balancing it, so that the values 25, 6, 9, 5, 99, 100, 101, 7 produce the tree:

```
          25
     6          99
5      9          100
     7                101
```

By traversing the tree in infix order — go left if possible, return value, go right if possible — the values are returned in sorted order. Instead of constructing the binary tree with each vertex having two pointers and a value, build the tree using a coroutine for each vertex. (A coroutine must be self-contained, i.e., it cannot access any global variables in the program.)

The coroutine has the following interface (you may only add a public destructor and private members):

```
template<typename T> _Coroutine Binsertsort {
    T value;                        // communication: value passed down/up tree
    void main();                    // YOU WRITE THIS ROUTINE
  public:
    _Exception Sentinel {};
    void sort( T value ) {          // value to be sorted
        Binsertsort::value = value;
        resume();
    }
    T retrieve() {                  // retrieve sorted value
        resume();
        return value;
    }
};
```

Assume type T has operators ==, <, >> and <<, and public default and copy constructors.

Each value for sorting is passed to the coroutine via member sort. When passed the first value, v, the coroutine stores it in a local variable, pivot. Each subsequent value is compared to pivot. If v < pivot, a Binsertsort coroutine called less is resumed with v; if v => pivot, a Binsertsort coroutine called greater is resumed with v. Each of the two coroutines, less and greater, creates two more coroutines in turn. The result is a binary tree of identical coroutines. The coroutines less and greater must be created on the stack not by calls to **new**, i.e., no dynamic allocation is necessary in this coroutine. Also, do not create coroutines unnecessarily. Let *leaf* mean a node that has received a pivot value but no further values. Ensure a leaf node does not contain left/right coroutines because both would be unused. However, unused coroutine nodes are allowed in non-leaf nodes.

The end of the set of values is indicated by raising the Sentinel exception at the root coroutine to start retrieval. The Sentinel exception indicates the end of unsorted values, and a coroutine that catches a Sentinel exception

raises a Sentinel exception at its left branch, prepares to receive the sorted values from its left branch, by calling its retrieve, passes these sorted values up the tree as the return value its retrieve call, and does so until it receives a Sentinel exception from the child coroutine on that branch. The coroutine then passes up its pivot value. Then the coroutine raises a Sentinel exception at its right branch, prepares to receive the sorted values from its right branch, passes these values up the tree as it did for the left branch, and does so until it receives a Sentinel exception from the child coroutine on that branch. Finally, the coroutine raises the Sentinel exception at its resumer to indicate the end of sorted values and terminates; hence, *all* coroutines must raise the Sentinel exception before terminating. (Note, the coroutine does **not** print out the sorted values — it simply returns them to its resumer.)

Handle a set of 0 or 1 values in the coroutine versus special cases in the program main, i.e., a Sentinel exception is raised as the first or second action to the sort coroutine.

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

```
binsertsort unsorted-file [ sorted-file ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.)

- If the unsorted input file is not specified, print an appropriate usage message and terminate. The input file contains lists of unsorted values. Each list starts with the number of values in that list. For example, the input file:

  ```
  8 25 6 8 5 99 100 101 7
  3 1 3 5
  0
  10 9 8 7 6 5 4 3 2 1 0
  ```

  contains 4 lists with 8, 3, 0 and 10 values in each list. (The line breaks are for readability only; values can be separated by any white-space character and appear across any number of lines.) **The input-value type is provided externally by the preprocessor variable BTYPE (see the Makefile).**

  Assume the first number of every line in the input file is always present and correctly specifies the number of following values. Assume all following values are correctly formed so no error checking is required on the input data.

- If no output file name is specified, use standard output. Print the original input list followed by the sorted list, as in:

  ```
  25 6 8 5 99 100 101 7
  5 6 7 8 25 99 100 101

  1 3 5
  1 3 5

  blank line from list of length 0 (not actually printed)
  blank line from list of length 0 (not actually printed)

  9 8 7 6 5 4 3 2 1 0
  0 1 2 3 4 5 6 7 8 9
  ```

  for the previous input file. End each set of output with a blank line.

Print an appropriate error message and terminate the program if unable to open the given files. See uIO.cc for an example of μC++ command-line parsing and file I/O.

Because Binsertsort is a template, show an example of it sorting a *non-basic* type, e.g., a structure with multiple values that provides operators ==, <, >> and <<, respectively. Include this example type in the same file as the program main. Note, string *is* a basic type and must be sortable by Binsertsort.

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 print( unsigned int id, int took, unsigned int RemainingPlayers );  // card play
    void print( 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

Last Card Taken			No Players Remaining			Last Card / Death			Drinks		
Players: 3		Cards: 46	Players: 3		Cards: 87	Players: 3		Cards: 59	Players: 3		Cards: 33
P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2
6:26>	8:38>	6:32>	1:84>	2:85<		5:44>X	6:49<	4:55<	4:27<	2:31<	7:20>
6:15<X	5:21<	7:8>	3:70>	6:78>X	5:73<		5:39<	6:33<	D	D	D
	2:6>	6:0#	#67#		3:67<X		8:25<	6:19<	1:19<	3:10>	6:13<
							4:15<	8:7<	2:4>	4:0#	4:6>
							7:0#X				

Figure 1: Schmilblick Card-Game Output

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.

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

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

static volatile long int shared = 0;          // volatile to prevent dead-code removal
static intmax_t iterations = 500000000;

_Task increment {
    void main() {
        for ( decltype(iterations) i = 0; i < iterations; i += 1 ) {
            shared += 1;                      // multiple increments to increase pipeline size
            shared += 1;
        } // for
    } // increment::main
}; // increment

int main( int argc, char * argv[] ) {
    intmax_t processors = 1;
    struct cmd_error {};
    try {                                     // process command-line arguments
        switch ( argc ) {
          case 3: processors = convert( argv[2] ); if ( processors <= 0 ) throw cmd_error{};
          case 2: iterations = convert( argv[1] ); if ( iterations <= 0 ) throw cmd_error{};
          case 1: break;                      // use defaults
          default: throw cmd_error{};
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ iterations (> 0) [ processors (> 0) ] ]" << endl;
        exit( EXIT_FAILURE );                 // TERMINATE!
    } // try

    uProcessor p[processors - 1];             // create additional kernel threads
    {
        increment t[processors == 1 ? 2 : processors];
    } // wait for tasks to finish
    cout << "shared:" << shared << endl;
} // main
```

Figure 2: Interference

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.

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 1000000000 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 1000000000 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 1000000000? Explain your answers. (**Hint:** one of the obvious answers is wrong.)

   (d) **BONUS:** 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. q1binsertsort.h, 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. q2*.{h,cc,C,cpp} – code for question 2, p. 3. **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.**

4. q3*.txt – contains the information required by question 3.

5. 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.

```
BTYPE := int
OPT := -O2

CXX = u++                                          # compiler
CXXFLAGS = -quiet -g -Wall -Wextra ${OPT} -MMD -DBTYPE="${BTYPE}"
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

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

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

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

-include BinsertImpl

ifeq (${IMPLBTYPE},${BTYPE})                       # same implementation type as last time ?
${EXEC1} : ${OBJECTS1}
    ${CXX} $^ -o $@
else                                               # implementation type has changed => rebuilt
.PHONY : ${EXEC1}
${EXEC1} :
    rm -f BinsertImpl
    touch q1binsertsort.h
    sleep 1
    ${MAKE} ${EXEC1} BTYPE="${BTYPE}"
endif
```

```
BinsertImpl :
    echo "IMPLBTYPE=${BTYPE}" > BinsertImpl
    sleep 1

${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 BinsertImpl ${EXECS}
```

This makefile is used as follows:

```
$ make binsertsort BTYPE=int
$ ./binsertsort intdata
$ make binsertsort BTYPE=double
$ ./binsertsort doubledata
$ make cardgame
$ ./cardgame ...
```

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