CS 343 Winter 2017 – Assignment 2
Instructor: Peter Buhr
Due Date: Sunday, October 8, 2017 at 22:00
Late Date: Thursday, October 12, 2017 at 22:00

September 29, 2017

This assignment has 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, i.e., 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 *semi-coroutine* to sort a set of 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. Hence, each coroutine in the tree contains two other coroutines and a value. (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 {
    const T Sentinel;            // value denoting end of set
    T value;                     // communication: value being passed down/up the tree
    void main();                 // YOU WRITE THIS ROUTINE
  public:
    Binsertsort( T Sentinel ) : Sentinel( 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 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.

   The end of the set of values is signaled by passing the value Sentinel; Sentinel is initialized when the sort coroutine is created via its constructor. (The Sentinel is not considered to be part of the set of values.) When a coroutine receives a value of Sentinel, it indicates end-of-unsorted-values, and the coroutine resumes its left branch (if it exists) with a value of Sentinel, prepares to receive the sorted values from the left branch and pass them back up the tree until it receives a value of Sentinel from that branch. The coroutine then passes up its pivot value. Next, the coroutine resumes its right branch (if it exists) with a value of Sentinel, prepares to receive the sorted values from the right branch and pass them back up the tree until it receives a value of Sentinel from that branch. Finally, the coroutine passes up a value of Sentinel to indicate end-of-sorted-values and the coroutine terminates. (Note, the coroutine does not print out the sorted values it simply passes them to its resumer.)

Remember to handle the special cases where a set of values is 0 or 1, e.g., Sentinel being passed as the first or second value to the coroutine. These cases must be handled by the coroutine versus special cases in the program main.

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.) The type of the input values and the sentinel value are specified externally by preprocessor variables TYPE and SENTINEL, respectively.

- 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.)

  Assume the first number 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, and none of the following values are the Sentinel value.

- 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.

Because Binsertsort is a template, show an example where it can sort any *non-basic* type (like a structure with multiple values forming the key) that provides operators ==, <, >> and <<, respectively. Include this example type in the same file as the program main.

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.

```
    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: Card Game Example Output

2. Write a *full coroutine* that plays the following card game. If a player is not the only player, they take a number of cards from a deck of cards 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. *After making a play*, a player checks to see if they received a deck that is a multiple of 7 ("death deck"); if so, that player must remove themself from the game. (A player always makes a play, otherwise 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.

   At random times, 1 in 10 plays, a player takes a drink and raises the Schmilblick *resumption* exception at the player on their right. In the handler for the Schmilblick exception of the player on the right, they take a drink and raise the Schmilblick resumption exception at the player on their right, and so on, until the handler is invoked for the player who started the Schmilblick. After all the players have had a drink, the game continues exactly where it left off. Note, a dead player 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( Printer &printer, unsigned int id );
    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. 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 routine also resumes the player coroutine to set the program main as its starter (needed during termination). 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, int RemainingPlayers );
};
```

   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 drinking or making a play "taken:remainingdirection":

a) the number of cards taken by a player,
b) the number of cards remaining in the deck,
c) the direction 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 a player to simultaneously receive a death-deck and win the game. In these cases, the output indicates both the win and the 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; **do not build and store strings of text for output**.

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 | "x" [ players | "x" [ cards | "x" [ seed | "x" ] ] ] ]
```

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

players is the number of players in the game (≥ 2). If no value for players is specified or x, 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 x, 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 x, 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 class PRNG. There are up to five calls to obtain random values in the program. Three calls in the main routine, depending on the command-line arguments, to obtain: 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). Two calls in Player to ~~start a Schmilblick and make a random play~~ make a random play and start a Schmilblick (in that order).

There are a couple of places in this question where a flag variable may be the best approach.

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.

3. Compile the program in Figure 2 using the u++ command, without and with compilation flag -multi and ***no optimization***, to generate a uniprocessor and multiprocessor executable. Run both versions of the program 10 times with command line argument 10000000 on a multi-core computer with at least 2 CPUs (cores).

   (a) Show the 10 results from each version of the program.

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

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

volatile int iterations = 10000000, shared = 0;    // ignore volatile, prevent dead-code removal

_Task increment {
    void main() {
        for ( int i = 1; i <= iterations; i += 1 ) {
            shared += 1;    // no -O2 to prevent atomic increment instruction
        }
    }
};
int main( int argc, char * argv[] ) {
    if ( argc == 2 ) iterations = atoi( argv[1] );
#ifdef __U_MULTI__
    uProcessor p;                          // create 2nd kernel thread
#endif // __U_MULTI__
    {
        increment t[2];
    } // wait for tasks to finish
    cout << "shared:" << shared << endl;
}
```

Figure 2: Interference

(c) In theory, what are the smallest and largest values that could be printed out by this program with an argument of 10000000? Explain your answers. (**Hint:** one of the obvious answers is wrong.)

(d) Explain the difference in the size of the values between the uniprocessor and multiprocessor output.

Submission Guidelines

Please follow these guidelines very carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text file, i.e., *.*txt file, must be ASCII text and not exceed 500 lines in length, where a line is a maximum of 120 characters.** 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. The file q1binsertsort.h contains the template code of the binary-tree insertion-sort. **Program documentation must be present in your submitted code. No user, system or test documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**

2. PRNG.h – random number generator (provided)

3. q2*.{h,cc,C,cpp} – code for question 2, p. 3. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**

4. q2*.testtxt – test documentation for question 2, p. 3, 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.**

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.

```
TYPE:=int
SENTINEL:=-1

CXX = u++                                        # compiler
CXXFLAGS = -g -Wall -MMD -DTYPE="${TYPE}" -DSENTINEL="${SENTINEL}"
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

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

OBJECTS2 = # object files forming 2st 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

all : ${EXECS}                                   # build all executables

#############################################################
-include ImplType

ifeq (${IMPLTYPE},${TYPE})                       # same implementation type as last time ?
${EXEC1} : ${OBJECTS1}
	${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${TYPE},)                                  # no implementation type specified ?
# set type to previous type
TYPE=${IMPLTYPE}
${EXEC1} : ${OBJECTS1}
	${CXX} ${CXXFLAGS} $^ -o $@
else                                             # implementation type has changed
.PHONY : ${EXEC1}
${EXEC1} :
	rm -f ImplType
	touch q1binsertsort.h
	sleep 1
	${MAKE} ${EXEC1} TYPE="${TYPE}" SENTINEL="${SENTINEL}"
endif
endif

ImplType :
	echo "IMPLTYPE=${TYPE}" > ImplType
	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 ${EXECS} ImplType
```

This makefile is used as follows:

```
$ make binsertsort
$ binsertsort ...
$ 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!