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 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     100 7
   /     /  \
   9     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):

```cpp
template<typename T> _Coroutine Binsertsort {
    T value;    // communication: value being passed down/up the tree
    void main();    // YOU WRITE THIS ROUTINE
public:
    _Event Sentinel {};    // value to be sorted
    void sort( T value ) {
        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. Also, do not create coroutines less and greater for a leaf node, i.e., at least one of coroutines less and greater must be used. Hence, the start of the coroutine looks similar to:
pivot = value;

try {
    suspend();               // get next value
} catch (Sentinel &) {
    // leaf retrieval finished
    // implies leaf node
}

// implies vertex node

Binsertsort<T> less, greater; // create less and greater

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, and passes these values up the tree 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, and passes these values up the tree 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, e.g., a Sentinel exception is raised as the first or second action to the sort coroutine, in 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 is specified externally by preprocessor variable TYPE.

- 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.

- 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 Binsort is a template, show an example where it can sort any non-basic type (string is a basic type; a structure with multiple values and a key is not a basic type) 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. 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 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 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):

```cpp
class Potato {
    // YOU ADD MEMBERS HERE
    public:
        _Event Explode {};  // Event raised when potato explodes
        Potato( unsigned int maxTicks = 10 );
        void reset( unsigned int maxTicks = 10 );
        void countdown();
};
```

The constructor is optionally passed 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. 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 is (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 global 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 as part of eliminating the umpire from the game after an election selects a new umpire.

The constructor is passed an assigned identification number (0 to \( N - 1 \)) and the potato created by the main program. The start member receives a players left and right partners. The toss member is called to conceptually pass the potato to another player. A player randomly tosses to the player on their left or right. The vote member is called to vote in an election and is passed the current highest player id seen during traversal of the circle. The terminate member is called to restart the umpire to propagate the terminate exception. Note, good software engineering encapsulates all calls to resume is interface members, rather than directly calling resume for another player.

If the umpire is eliminated, i.e., the timer went off while holding the potato, it starts an election. After the election finishes, the old umpire raises the Terminate exception at the new umpire and calls terminate; the call to terminate should never return.
Coroutine Player {
  _Event Terminate {
    public:
    Player & victim; // delete player
    Terminate( Player & victim ) : victim( victim ) {} 
  };
  _Event Election {
    public:
    unsigned int id; // highest id seen so far
    Election( unsigned int id ) : id( id ) {} 
  };
  Player * partner[2]; // left and right player
  // YOU ADD MEMBERS HERE
  void main();
  void vote( unsigned int id ); // resume partner to vote
  void terminate(); // resume umpire
  public:
  static Player * umpire; // current umpire
  Player( unsigned int id, Potato & potato );
  void start( Player & lp, Player & rp ); // supply partners
  void toss(); // tossed the potato
};

Figure 1: Player Interface

The new umpire handles the Terminate exception, deleting the eliminated player. Hence, player coroutines are created in the main program but deleted during the game by the umpire. The new umpire starts a new game by unlinking the terminated player from the circle, deleting it, resetting the potato timer, and the game begins with the umpire randomly tossing the potato to its left/right player. This toss counts with respect to the timer in the potato.

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.

The program main creates the potato and players in increasing order of player id. After creating the players, generate a random player index and swap that position with position 0, hence players are not in increasing order of 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 routine also resumes the player to set the program main as its starter (needed during termination). The main routine 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, 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.

To obtain repeatable results, all random numbers are generated using class PRNG. There are up to four calls to this random-number generator: 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.

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
$ hotpotato 1 6 1003
6 players in the game
POTATO goes off after 8 tosses
U 0 -> 5 -> 0 -> 5 -> 0 -> 5 -> 0 -> 5 is eliminated
POTATO goes off after 7 tosses
U 0 -> 3 -> 2 -> 1 -> 4 -> 1 -> 4 is eliminated
POTATO goes off after 7 tosses
U 0 -> 3 -> 2 -> 3 -> 2 -> 3 -> 2 is eliminated
POTATO goes off after 8 tosses
U 0 -> 3 -> 1 -> 3 -> 0 -> 3 -> 1 -> 3 is eliminated
POTATO goes off after 7 tosses
U 0 -> 1 -> 0 -> 1 -> 0 -> 1 -> 0 is eliminated
E 0 -> 1 : umpire 1
POTATO goes off after 1 toss
U 1 wins the Match!

Figure 2: Sample Output

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 3 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 100000000 on a multi-core computer with at least 2 CPUs (cores).

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

volatile long int iterations = 100000000, shared = 0; // volatile to prevent dead-code removal

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

int main( int argc, char * argv[ ] ) {
    try {
        // process command-line arguments
        switch ( argc ) {
            case 2: iterations = stoi( argv[1] );
            case 1: break;
            default: throw 1;
        } // switch
        catch( . . . ) {
            cout << "Usage: " << argv[0] << " [ iterations (> 0) ]" << endl;
            exit( 1 );
        } // try
        #ifdef __U_MULTI__
        uProcessor p; // create 2nd kernel thread
        #endif // __U_MULTI__
        { increment t[2];
            // wait for tasks to finish
            cout << "shared:" << shared << endl;
        }
    }
```
(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.
(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 the difference in the size of the values between the uniprocessor and multiprocessor output.

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, doc file, must be ASCII text and not exceed 500 lines in length, using the command fold -w120 *.doc | 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. q1binsort.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*.doc – 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. 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.

```
TYPE := int
CXX := u++
CXXFLAGS := -g -Wall -Wextra -MMD -DTYPE="${TYPE}"
MAKEFILE_NAME := $(firstword $(MAKEFILE_LIST)) # makefile name

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

OBJECTS2 = # object files forming 2st executable with prefix "q2"
EXEC2 = hotpotato

OBJECTS = $(OBJECTS1) $(OBJECTS2) # all object files
DEPENDS = $(OBJECTS:.o=.d) # substitute ".o" with ".d"
EXECES = $(EXEC1) $(EXEC2) # all executables

.PHONY : all clean

all : $(EXECES) # build all executables
```

```
```
`
-include ImplType

ifeq (${IMPL TYPE},${TYPE}) 
$${EXEC1} : $${OBJECTS1} 
  $${CXX} $${CXXFLAGS} $^ -o $$@ 
else
ifeq ($${TYPE},)
  # no implementation type specified ?
 (TYPE)=${IMPL TYPE} 
else
# implementation type has changed

.PHONY : $${EXEC1}
$$({EXEC1}) :
  rm -f ImplType
  touch q1binsortsor.h
  sleep 1
  $(MAKE) $$({EXEC1}) TYPE="$${TYPE}" 
endif
endif

ImplType :
  echo "IMPLTYPE=${TYPE}" > ImplType
  sleep 1

ifeq ($${EXEC2},${OBJECTS2}) 
  $${CXX} $${CXXFLAGS} $^ -o $$@ 
else
# link step 2nd executable

$(OBJECTS) : $(MAKEFILE_NAME) 
-include $(DEPENDS) 
# include *.d files containing program dependences

clean :
  rm -f *.d *.o $(EXECS) ImplType

This makefile is used as follows:

$ make binsortsor
$ binsortsor ... 
$ make hotpotato
$ hotpotato ...

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