CS 246 Winter 2016 – Assignment 4
Instructors: Peter Buhr and Rob Schluntz
Due Date: Monday, March 21, 2016 at 22:00

March 16, 2016

This assignment studies advance C++, inheritance, and design patterns. Use it to become familiar with these facilities, and ensure you use the specified 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.)

Do one of questions 1 or 2. (Question 1 has two parts.) Each of these questions leads into a project question for assignment 5. Question 1 leads into the CC3K project, while question 2 leads into the WATCola project. Read over the project questions to determine which question you want to do in assignment 4.

0. Start looking for a partner for assignment 5. Once you have a partner, only one partner submits both partner names to Marmoset in file partner.txt, e.g.:

   ```
   userid1
   userid2
   ```

   where userid1 and userid2 are UW userids, e.g. jfdoe. The project may be done individually, but it must be understood that the amount of work is significantly greater and no extra marks are given for this additional work. Do **NOT** submit a partner.txt file, if you plan to do the project individually.

1. (a) Write a program to read and evaluate arithmetic expressions. There are three kinds of expressions:

   - integer constant
   - unary operator (NEG or ABS, denoting negation and absolute value) applied to an expression
   - binary operator (+, -, *, or /) applied to two expressions

   Expressions are entered in reverse Polish notation (RPN), also known as postfix notation, in which the operator is written after its operands. For example, the input

   ```
   12 34 7 + * NEG
   ```

   denotes the expression

   ```
   (-(12 * (34 + 7))
   ```

   The program reads in an expression, prints its value in conventional infix notation, and then computes the expression and prints the expression's value. For example (output in italics):

   ```
   1 2 + 3 4 - * ABS NEG
   -|((1 + 2) * (3 - 4))|
   = -3
   ```

   To solve this question, define a base class Expression, and a derived class for each of the the three kinds of expressions, as outlined above. The base class provides virtual methods prettyprint and evaluate that carry out the required tasks.

   A stack is needed to read an RPN expression. Use cin with operator >> to read the input one word at a time. If the word is a number, create a corresponding expression object, and push a pointer to the object onto the stack. If the word is an operator, pop one or two items from the stack according to the kind of operator (unary or binary), convert to the corresponding object, and push back onto the stack. When the input is exhausted, the stack contains a pointer to a single object that encapsulates the entire expression.

   For the stack, use an array of pointers to expression objects. The array should have size 10. If at any point more than this amount of stack space is required, print "Stack overflow" to cerr and terminate the program. Otherwise, you may assume that you are given valid input.

Once you have read in the expression, print it out in infix notation with full parenthesization, as illustrated above. Then evaluate the expression and print the result.

Note: The design used in this question is an example of the *interpreter pattern*.

(b) Write a set of C++ classes to implement the game of Flood It, a one player game. You can play a graphical version of the game at http://unixpapa.com/floodit. An instance of Flood consists of an $n \times n$-grid of cells, each of which can be in one of 5 states, 0, 1, 2, 3, or 4 (with the default state being 0). Before the game begins, an initial configuration of the state cells will be randomly generated. Once the cells are configured, the player repeatedly changes the state of the top-left (0,0) cell. In response, all neighbouring cells (to the north, south, east, and west) switch state if they were in the same state as a neighbouring cell that changed states. The action is referred to as flooding. The object of the game is to have all cells in the grid be in the same state before running out of moves. To implement the game, use the following classes:

- Cell - implements a single cell in the grid (see provided cell.h)
- Game - contains the grid of cells and implements the game logic (see game.h)
- TextDisplay - responsible for displaying the text version of the grid.

The Game contains a grid of Cells. Each Cell is an observer of its neighbours (which means class Cell is its own observer)[1]. Game calls Cell::notify on a given Cell and ask it to change state. Note, because of the way the game is played, it only makes sense for Game to call Cell::notify on the (0,0) cell with a single parameter, the new state of the cell. The notified cell must then call a notify method on each of its neighbours (each cell is told who its neighbours are when the grid is initialized). In this notification, you may find it useful to send the Cell's current and previous states as parameters (to prevent infinite notification loops). Each time a Cell changes state, it must notify TextDisplay of its new state.

Every time the state of a Cell is updated, it sends an update to the Game which sends an update to the TextDisplay. Calling TextDisplay::print prints the grid to the screen (see example below).

The program reads commands from standard input. The program will read a sequence of intgers between 0 and 4 inclusive. For each integer n which is read, the board is flooded as described above.

The game is lost if the board is not in one state within g moves. You may assume that inputs are valid and command line options are valid. If the game is won, the program should display Won to standard output before terminating; if the game is lost, it should display Lost. The program ends when the input stream is exhausted or when the game is won or lost. If input was exhausted before the game was won or lost, it should display nothing.

The executable program is named floodit and has the following shell options:

```
floodit [-size n] | [-moves m] | [-seed val]
```

These command line options can be present in any order. n is the size of the grid and must be an integer which is at least 2. If no value for size is specified, use the value 10. m is the number of moves until the the game is over and must be at least 1. If no value for moves is specified, use the value 20. val is the value that is used to initialize the pseudo random number generator. If no value for seed is specified, the random number generator is initialized with the program process-id (see getpid), so that each run of the program generates different output. All arguments must be checked for correct type and range.

To obtain repeatable results, all random numbers are generated using class PRNG, which is created as a global variable, initialized, and exported in the main translation-unit to other translation units. To ensure consistency (for testing purposes), initialize the cells in order from row 0 to row $n-1$. Within a row, initialize each cell from the 0th index to the $(n-1)th$ index. Generating the state of each cell is done by the call `prng( 4 )`.

Figure 1 shows a possible game run using the command `floodit -moves 4 -size 4` (responses from the user are underlined).

2. Simulate a game of Hot Potato. The game consists of an umpire and a number of different kinds of players. The umpire starts a *set* by tossing a particular kind of *hot* potato to a player. What makes the potato *hot* is the timer inside it. The potato is then tossed among the players until the timer goes off. A player can toss the potato to themself. The player holding the potato when the timer goes off is eliminated and returns its id back

[1] Seek additional resources about the observer pattern. (Try the internet.)

```
4032
0103
4021
0203
4 moves left
0
0032
0103
4021
0203
3 moves left
4
4432
4103
4021
0203
2 moves left
0
0032
0103
0021
0203
1 move left
2
2232
2103
2221
2203
0 moves left
Lost
```

Figure 1: Floodit : Sample Game

to the player that tossed it the potato. That player returns, and so on, until control returns to the umpire with the terminated player as the return value. The umpire then shortens the list of players, selects one of the potatoes, and begins the next *set* with the remaining players, unless only one player remains, which the umpire declares the winner and the game is over.

For the following interfaces, you may NOT change or remove from any of the GIVEN interface declarations or add any additional public declarations, except a virtual public destructor if needed. You may add private/protected members.

The potato is the item tossed around by the players and it also contains the timer that goes off after a period of time. All potatoes are derived from the following abstract interface:

```
class Potato {
    // YOU ADD MEMBERS
  public:
    struct Expire {};                    // raise when timer expires
    Potato( Printer &prt );
    virtual void reset() = 0;            // must be defined in derived class
    virtual void countdown();
};
```

There are two kinds of potatoes:

```
class Mashed : public Potato {
    // YOU ADD MEMBERS
  public:
    Mashed( Printer &prt, unsigned int maxTicks = 10 );
    void reset();
    void countdown();
};
```

```
class Fried : public Potato {
    // YOU ADD MEMBERS
  public:
    Fried( Printer &prt, unsigned int maxTicks = 10 );
    void reset();
    void countdown();
};
```

The constructor is optionally passed the maximum number of ticks until the timer goes off. A mashed potato chooses a random value in the range [1,maxTicks] for the number of ticks. A fried potato always chooses the given maximum number of ticks. Member reset is called by the umpire to re-initialize the timer to reuse the potato. When a potato (re)sets its timer, it prints a message indicating the number of ticks before going off. Member countdown is called by each player, and it decrements the timer and raises exception Expire if the timer has gone off. Rather than use absolute time to implement the potato's timer, make each call to countdown be one tick of the clock. To reduce code duplication, factor as much common code between the two kinds of potatoes into the base class Potato.

All players must be derived from the following abstract interface:

```
class Player {
    // YOU ADD MEMBERS
  public:
    typedef ... Players;             // container type of your choice
    struct Lost {                    // raise after timer expires
        // YOU ADD MEMBERS
    };
    Player( Printer &prt, unsigned int id, Players &players );
    virtual unsigned int getId();
    virtual void toss( Potato &potato ) = 0; // must be defined in derived class
};
```

The type Players is a container of your choice (e.g., array or any of the C++ standard containers), containing all the players still in the game. Member getId returns the player id. There are two kinds of players:

```
class RNPlayer : public Player {
    // YOU ADD MEMBERS
  public:
    RNPlayer( Printer &prt, unsigned int id, Players &players );
    void toss( Potato &potato );
};
class LRPlayer : public Player {
    // YOU ADD MEMBERS
  public:
    LRPlayer( Printer &prt, unsigned int id, Players &players );
    void toss( Potato &potato );
};
```

The constructor is passed an identification number (id) assigned by the main program and the container of players. If a player is not eliminated while holding the *hot* potato, then the player chooses another player from the list of players and tosses it the potato using its toss member. RNPlayer players choose any random player still in the game to receive the potato. LRPlayer players alternate choosing the player on the left or right (modulo the list size), starting with the player on the left. Member toss raises exception Lost when the timer expire for the player holding the potato. Return the id of the player to be eliminated in the exception Lost.

The interface for the Umpire is:

```
class Umpire {
    // YOU ADD MEMBERS
  public:
    Umpire( Printer &prt, unsigned int maxticks, Player::Players &players );
    void start();
};
```

```
$ hotpotato 6 20 1003
M       F       U       P0      P1      P2      P3      P4      P5
===     ===     ===     ===     ===     ===     ===     ===     ===
8       20      M 4     r 5     R 0             R 1     r 3     R 5
                                        r 1                     R 2
                        ...     *       ...     ...     ...     ...
        20      F 2                     l 3     R 4     l 5     R 4
                        l 2             r 0     R 2     r 3
                        r 5             l 3     R 0             R 3
                        l 2             r 0     R 0
                        r 5             l 3     R 5             R 2
                        ...     ...     ...     ...     ...     *
1               M 0
                        *       ...     ...     ...     ...     ...
        20      F 3                     r 4     R 2     l 2
                                        l 3     R 4     r 3
                                                R 3
                                                R 3
                                        r 4     R 2     l 2
                                        l 3     R 4     r 3
                                                R 3
                                        r 4     R 4     l 2
                                                        r 3
                        ...     ...     ...     *       ...     ...
8               M 4                     l 4             l 2
                                        r 4             r 2
                                        l 4             l 2
                                                        r 2
                        ...     ...     *       ...     ...     ...
                W 4
```

Figure 2: Hotpotato Example Output

The constructor is passed the container with the players in the game. start is called by the main program to begin a game. The umpire creates a mashed and fried potato and initializes them both with maxticks. Then it alternates choosing between the two kinds of potato (starting with mashed) and tosses it to a randomly selected player to start the next set; this toss counts with respect to the timer in the potato. When a player determines it is eliminated, i.e., the timer went off while holding the potato, it returns its id, which is eventually returned to the umpire. The umpire removes the eliminated player from the container of players and deletes the storage for the player. Removal of a player must maintain the relative position of the other players in the container. The umpire then chooses the other kind of potato, resets that potato, and tosses it to a randomly selected player to start the next set.

All output from the program is generated by calls to a printer, excluding error messages.

```
class Printer {
  public:
    enum Kind { Mashed, Fried, Umpire, Player };
    Printer( unsigned int players );
    ~Printer();
    void print( Kind kind, unsigned int state, unsigned int id = 0, unsigned int player = 0 );
};
```

The printer generates output like that in Figure 2, which is a dynamic display of the game. Each column is assigned to a particular kind of object. There are 4 kinds of objects: Mashed, Fried, Umpire, and Player, where Players have multiple instances. An object with multiple instances passes in its local identifier (id) [0,N) when printing. Each kind of object prints specific information in its column:

- Mashed and Fried print the number of ticks before the timer goes off.
- Umpire prints the kind of potato used and which player the potato is tossed to start the game, and W and the player who wins the game at the end.

- Player prints r (right) or l left for the toss direction of an LRPlayer player and R for a RNPlayer, followed by the player the potato is tossed to. When a player terminates, a `'*'` is printed in the terminated player's column and all other player columns print "...".

The only tricky aspect of the printer is not printing unnecessary tabs after the right most column in a row. Hint: keep track of the number of items in the buffer and use this value to decide when to stop printing the elements in a row.

To condense the output, information is buffered until a column is overwritten for a particular entry, which causes the buffered data to be flushed. If there is no new stored information for a column since the last buffer flush, an empty column is printed. After a task has finished, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing. Buffer any information necessary for printing in its internal representation; **do not build and store strings of text for output**.

Figure 3 shows a smaller run of the simulation with an execution trace.

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

```
hotpotato [ players [ maxticks [ seed ] ] ]
```

players is the number of players in the game and must be between 2 and 20, inclusive. If no value for players is specified, use the value 5. maxticks is the initialization value for a potato and must be great than 1. If no value for maxticks is specified, use the value 10. seed is the seed for the random number generator, in the range [1. .N], so it is possible to generate repeatable output. If no value for seed is specified, the random number generator is initialized with the program process-id (see getpid), so that each run of the program generates different output. All arguments must be checked for correct type and range. The main program creates the umpire, the players, alternating with either a LRPlayer or RNPlayer (starting with an LRPlayer) and player identifiers from 0 to players–1, and the container holding all the players. It then begins a game by calling Umpire::start.

To obtain repeatable results, all random numbers are generated using class PRNG, which is created as a global variable, initialized, and exported in the main translation-unit to other translation units. There are three calls to obtain random values in the program: one in Mashed::reset, one in RNPlayer::toss, and one in Umpire::start. The number of calls to the random-number generator must be the same as the sample executable to obtain identical output.

Submission Guidelines

Please follow these guidelines 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.** Name your submitted files as follows:

1. (a) rpn.{cc,C,cpp}, {binary,expr,num,unary}.{h,cc,C,cpp} – code for question 1a, p. 1. The program must be divided into separate compilation units, i.e., .h and *.{cc,C,cpp} files. **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.**

   (b) PRNG.h, floodit.{cc,C,cpp}, {cell,game,textdisplay}.{h,cc,C,cpp} – code for question 1b, p. 2. **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.**

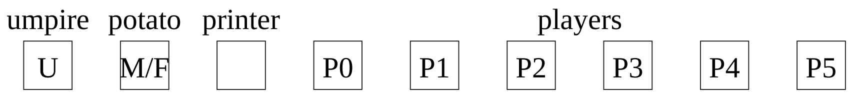
   floodit.testtxt – test documentation for question 2, p. 2, which includes the input and output of your tests. Write a brief description for each test explaining what aspects of the program it is testing and how you decided if the program passed the test.

2. PRNG.h, hotpotato.{cc,C,cpp}, {printer,player,potato,umpire}.{h,cc,C,cpp} – code for question 2, p. 2. The program must be divided into separate compilation units, i.e., .h and *.{cc,C,cpp} files. **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.**

   hotpotato.testtxt – test documentation for question 2, p. 2, which includes the input and output of your tests. Write a brief description for each test explaining what aspects of the program it is testing and how you decided if the program passed the test.

```
$ hotpotato 6 8 1003
M       F       U       P0      P1      P2      P3      P4      P5
===     ===     ===     ===     ===     ===     ===     ===     ===
4       8       M 4             R 0             R 1     r 3
                        *       ...     ...     ...     ...     ...
        8       F 3             R 4     r 1     R 5     l 5     R 2
                                                        r 3     R 4
                        ...     ...     ...     *       ...     ...
5               M 1             R 1
                                R 4                     l 5     R 4
                        ...     ...     ...     ...     *       ...
        8       F 5             R 5                             R 1
                                        l 5                     R 2
                                R 5                             R 1
                                                                R 2
                        ...     ...     *       ...     ...     ...
8               M 1             R 5                             R 1
                                R 1
                                R 5                             R 5
                                R 1                             R 1
                        ...     *       ...     ...     ...     ...
                W 5
```

Setup 6 players, 8 maximum ticks, 1003 random seed



Game

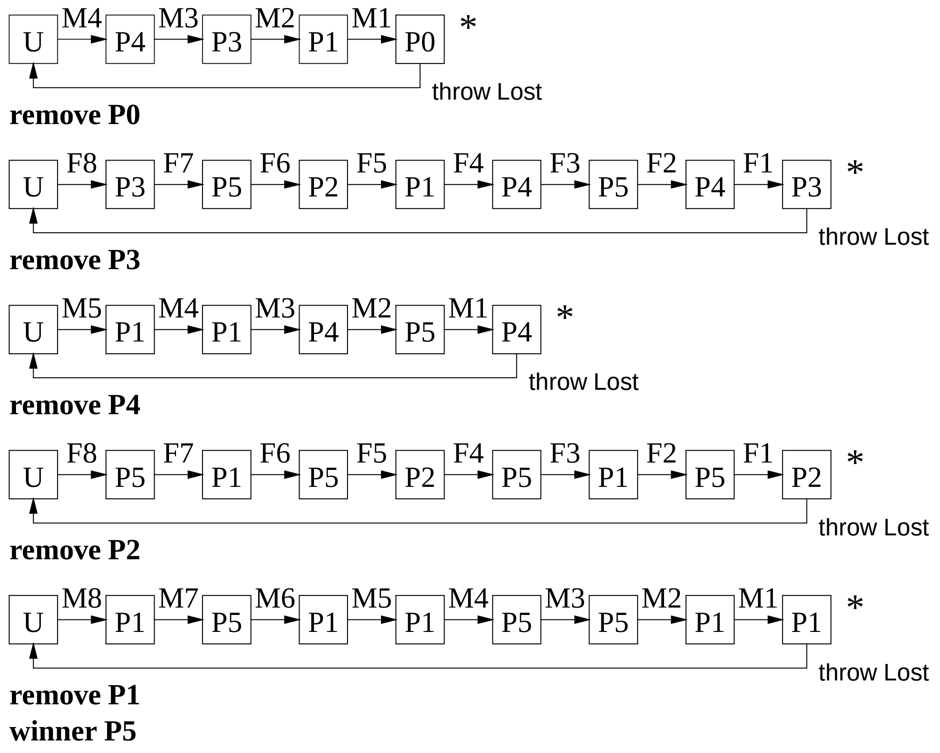

U M4 P4 M3 P3 M2 P1 M1 P0 *
throw Lost
remove P0
U F8 P3 F7 P5 F6 P2 F5 P1 F4 P4 F3 P5 F2 P4 F1 P3 *
throw Lost
remove P3
U M5 P1 M4 P1 M3 P4 M2 P5 M1 P4 *
throw Lost
remove P4
U F8 P5 F7 P1 F6 P5 F5 P2 F4 P5 F3 P1 F2 P5 F1 P2 *
throw Lost
remove P2
U M8 P1 M7 P5 M6 P1 M5 P1 M4 P5 M3 P5 M2 P1 M1 P1 *
throw Lost
remove P1
winner P5

Figure 3: Hotpotato Game Trace

Use the following Makefile to compile the programs for questions 1a, p. 1 and questions 1b, p. 2, or 2, p. 2:

```
CXX = g++-4.9                                        # compiler
CXXFLAGS = -g -Wall -Werror -std=c++11 -MMD          # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}}        # makefile name

OBJECTS1 = rpn.o binary.o expr.o num.o unary.o       # object files forming executable
EXEC1 = rpn                                          # executable name

OBJECTS2 = floodit.o cell.o game.o textdisplay.o     # object files forming executable
EXEC2 = floodit                                      # executable name

OBJECTS3 = hotpotato.o printer.o player.o potato.o umpire.o
EXEC3 = hotpotato                                    # executable name

OBJECTS = ${OBJECTS1} ${OBJECTS2} ${OBJECTS3}
EXECS = ${EXEC1} ${EXEC2} ${EXEC3}
DEPENDS = ${OBJECTS:.o=.d}                           # substitute ".o" with ".d"

.PHONY : all clean

all : ${EXECS}

${EXEC1} : ${OBJECTS1}                               # link step
	${CXX} $^ -o $@

${EXEC2} : ${OBJECTS2}                               # link step
	${CXX} $^ -o $@

${EXEC3} : ${OBJECTS3}                               # link step
	${CXX} $^ -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 ${DEPENDS} ${OBJECTS} ${EXECS}
```

Put this Makefile in the directory with your programs, name your source files appropriately, and then execute shell command make rpn or make floodit or make hotpotato in the directory to compile a program (make without an argument compiles all the programs). This Makefile is used by Marmoset to build programs, so make sure your programs compiles with it. *Do not make any changes to the* Makefile.

Follow these guidelines. Your grade depends on it!