CS-2303, System Programming Concepts, C-term 2017

Project 1 (20 points)
Assigned: Thursday, January 12, 2017
Due: Saturday, January 21, 2017, 6:00 PM

Programming Assignment #1 — Display a 12-month Calendar

Abstract

Write a *C* program called **`PA1.c`** that displays a twelve month calendar for an arbitrary year. Prompt the user for the year of the calendar, and print out the calendar month by month, so that it looks like a real calendar.

Outcomes

After successfully completing this assignment, you should be able to:–

- Develop a *C* program on a Unix or Linux platform
- Design a program that contains nested selection and iteration constructs and separate functions
- Specify the loop invariants that you use to reason about your program
- Use advanced formatting strings and conversion specifiers to do I/O in a *C* program

Before Starting

Re-read Chapter 1, Chapter 2, and sections §§3.1-3.5 K&R. These chapters should be very easy because of the similarity to Java. It is also suggested that you complete *Lab #1*, either during the scheduled lab session or during your own time.

The Assignment

Write a *C* program that displays twelve-month calendar for a particular year. The program should prompt the user for the year to be printed, and then it should figure out (a) whether the year is a leap-year and (b) what day of the week the chosen year starts on.

The calendar should be formatted as shown in the sample execution below. Note that numbers the days must be right-justified under the names of the days and that two spaces separate the names of the days from each other.

Interfaces

The interface **<stdio.h>** provides the functions **printf** and **scanf**.

Assumptions and Restrictions

The user may enter any positive integer for the year. You must calculate the calendar according to the modern international standard calendar (introduced by Pope Gregory XIII in the year 1582). For input years earlier than 1582, calculate them as if the modern calendar were in effect. In the modern calendar, years that are divisible by 4 are leap years, except that years divisible by 100 are not leap years unless they are also divisible by 400. That is, there are 97 leap years every four centuries.

You will have to figure out which day of the week the calendar starts on. You may do this by referring to a known year in which you know the day of the week of a particular date. You will then work backwards from that known date to find the start of the input year.[1]

Sample Execution

```
                    MONTHLY CALENDAR

Please enter year for this calendar:- 2009


  ***    CALENDAR for 2009   ***

January 2009

Sun  Mon  Tue  Wed  Thu  Fri  Sat
                    1    2    3
  4    5    6    7    8    9   10
 11   12   13   14   15   16   17
 18   19   20   21   22   23   24
 25   26   27   28   29   30   31

February 2009

Sun  Mon  Tue  Wed  Thu  Fri  Sat
  1    2    3    4    5    6    7
  8    9   10   11   12   13   14
                   .
                   .
                   .
```

(Output continues for all 12 months. Note that dates must be *right justified* under day names.)

[1] Little known fact:– The number of days in four centuries is exactly divisible by seven. This means that each four-century interval starts on the same day of the week.

Implementation Notes

Since we have not yet studied arrays, strings, or arrays of strings, you should design your algorithm to use **`if-else`** or **`switch`** statements to print the month names and to set other variables.

You should partition your program into at least three functions. Here is an example partition:–

- The function **`main()`** prompts the user for input, calls a function of your own design to determine the starting day of the input year. It then invokes the function **`printCalendar()`** to actually print the twelve month calendar.
- The function **`printCalendar()`** takes two arguments, the year number and the starting day. It then loops through the year and calls the function **`printMonth()`** twelve times, once for each month.
- The function **`printMonth()`** takes three arguments, the year number, the month number and the starting day of that particular month, and it returns the number of the day on which the next month starts. Print month has to first call a function **`printMonthName()`** and then print out the days of the month in calendar format.
- The function **`printMonthName()`** takes the year number and the month number as arguments, prints out the line identifying the month, and returns the number of days in that month, taking into account leap year. The example output of **`printMonthName()`** should look resemble the following:–

  `January 2009`

  Since we are not using arrays of strings, **`printMonthName()`** should use a **`switch`** statement to select and print the name of the month and to determine the number of days in that month. If the month is February, it should also figure out whether the year is a leap year and return the correct number of days.

Algorithm and Loop Invariants

There are many sources on the web and at WPI for a suitable algorithm for this assignment. You may consult any of these, *but you must cite your source*. If you worked out the algorithm on your own, you should say in your write up file that this is entirely your work.

Note: If you borrow some or all of an algorithm from someone else or from somewhere else, *do not copy it*. Write it out in your own words and your own coding style. Also, please explain enough about how the algorithm works that the graders can conclude that you understand it.

This project requires at least two loops. For each loop, write a *loop invariant* — that is, a logical statement in English or mathematical notation that says what salient facts are true about the relationships of the variables at the same point in the loop for each iteration.

Write each loop invariant as a comment in the body of the loop at the exact point during the execution of the loop body where the invariant is **`TRUE`**. Also include copy each loop invariant into your **`README`** document below.

Deliverables

Write a document called **`README.txt`**, **`README.doc`**, or **`README.docx`** summarizing your program, how to run it, and detailing any problems that you had. Also, if you borrowed all or part of

the algorithm for this assignment, be sure to cite your sources *and* explain in detail how it works. Be sure also to describe the *loop invariant* of each loop.

From browser, submit *C* source code file and **`README`** to *Canvas*.

Note: This course is listed as two separate lecture "courses" plus one common "Labs course." You are registered for one of the Lecture courses and also for the Labs course. This, along with all other programming assignments must be submitted to the *Assignments* section of the "Labs" course.

Programs submitted after the due date (Saturday, January 21, 2017, 6:00 PM) will be tagged as late, and will be subject to the late homework policy.

Grading

Graders will compile your assignment by executing the following command on an *Ubuntu* Linux system compatible with your course virtual machine:–

```
gcc -Wall -o PA1 PA1.c
```

Your program must *compile without errors in order to receive* any *credit* for this assignment. You *must not* use any extra switches — for example, **`-ansi`** or **`-std=C99`**. These will cause incompatibilities that will cause it to fail to compile for the graders. If you do your work on some other system, you may find that your system adheres to a slightly different standard and that some details of the *C* language may be different from those on the course virtual machine. Before submitting your assignment, be sure that it compiles on the course virtual and correct it if it does not.

This assignment is worth twenty (20) points:–

- Correct compilation without warnings – 2 points
- Correct execution with graders' test cases – 2 points
- Correct usage of **`scanf()`** to get inputs from user – 1 point
- Correct usage of **`print()`** to print the various lines of the calendar – 3 points
- Correct usage of conditional and loop statements – 5 points
- Satisfactory **`README`** file – 2 points
- Loop invariant for each loop in comments in the code *and also* in the **`README`** document – 5 points

CS-2303, System Programming Concepts, C-term 2017

Project 2 (40 points)
Assigned: Friday, January 20, 2017
Due: Saturday, January 28, 2017, 6:00 PM

Programming Assignment #1 — Game of Life

Abstract

Write a *C* program that plays the *Game of Life*. Accept as arguments the size of the board, the initial configuration, and the number of generations to play. Play that number of generations and display the final configuration of the board.

Outcomes

After successfully completing this assignment, you should be able to:–

- Develop a *C* program that uses two-dimensional arrays
- Allocate memory for the arrays at run time
- Pass these arrays as arguments to functions

Before Starting

Read Chapter 5 K&R pertaining to arrays and sections §§7.5–7.7 regarding file access. Pay particular attention to §5.10 about command line access and §§5.7–5.9 about multi-dimensional arrays.

John Conway's Game of Life

The Game of Life was invented by the mathematician John Conway and was originally described in the April 1970 issue of *Scientific American* (page 120). The Game of Life has since become an interesting object of mathematical study and amusement, and it is the subject of many websites.

The game is played on a rectangular grid of cells, so that each cell has eight neighbors (adjacent cells). Each cell is either occupied by an organism or not. A pattern of occupied and unoccupied cells in the grid is called a *generation*. The rules for deriving a new generation from the previous generation are these:–

1. *Death*. If an *occupied* cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, the organism dies (0 or 1 of loneliness; 4 thru 8 of overcrowding).

2. *Survival.* If an occupied cell has two or three neighbors, the organism survives to the next generation.
3. *Birth.* If an unoccupied cell has precisely three occupied neighbors, it becomes occupied by a new organism.

Examples can be found at http://www.math.com/students/wonders/life/life.html.

Once started with an initial configuration of organisms (Generation 0), the game continues from one generation to the next until one of three conditions is met for termination:

1. all organisms die, or
4. the pattern of organisms repeats itself from a previous generation, or
5. a predefined number of generations is reached.

Note that for some patterns, a new generation is identical to the previous one — i.e., a steady state. When this occurs, termination under condition #2 occurs. In some other common cases, a new generation is identical to the second previous generation; that is, the board oscillates back and forth between to two configurations. In rare cases, a pattern repeats after an interval of more than two generations. In still other cases (some not so rare), a pattern replaces itself by a fixed offset in one or both dimensions, thereby "flying" off the screen. In this assignment, you will be responsible for terminating after a steady state is reached or an oscillation of two alternating patterns is reached.

In theory, the Game of Life is played on an infinite grid. In this assignment, your program will play on a finite grid. The same rules apply, but squares beyond the edge of the grid are assumed to be always unoccupied.

Implementing your program

Your program should be called **`life`**. It needs to do several things:–

- Read the arguments to program from the command line.
- Read the initial configuration of the board from an **`input`** file.
- Allocate at least three arrays, each large enough to hold one generation of the game. Initialize the first generation with the initial configuration in the approximate center of the board.
- Play the game for as many generations as needed until one of the termination conditions above is met.
- Print out the final configuration, along with a message saying how many generations were played and under what condition the game terminated.

Program Arguments and Input

The program should be invoked from the command line with the following arguments:–

```
./life X Y gens input print pause
```

where

- **`X`** and **`Y`** are unsigned integers indicating the number of elements in the *x* and *y* directions if the grid, respectively.

- **`gens`** is the number of generations to play. This value must be greater than zero. The program should halt prior to this number of generations if it determines that the game has reached a termination condition.
- **`input`** is the name of a file containing a sequence of lines, each consisting of a sequence of **`'x'`** and **`'o'`** characters, indicating the occupied and unoccupied cells of the initial configuration.
- **`print`** is an optional argument with value of **`'y'`** or **`'n'`** indicating whether each generation (including generation 0) should be printed or displayed before proceeding to the next generation. If this item is missing, it defaults to **`'n'`.**
- **`pause`** is an optional argument with value of **`'y'`** or **`'n'`** indicating whether a keystroke is needed between generations. If this and/or the **`print`** item is missing, it defaults to **`'n'`.**

After interpreting the program arguments, your program must open the **`input`** file, read its lines, and initialize the configuration in the approximate center of your board in the *x*- and *y*-dimensions.

Example patterns

Here is a simple pattern that happens to be a "still life" or steady state:–

```
xx
xx
```

That is, the next generation starting from this pattern produces exactly the same pattern. Here is another still life pattern:–

```
oxo
xox
xox
oxo
```

The following pattern produces an oscillation between a vertical line of three occupied cells and a horizontal line of three occupied cells

```
x
x
x
```

The following pattern is a well-studied one called the *R-Pentomino*.

```
oxx
xxo
oxo
```

It creates an interesting sequence of generations, including many sub-patterns that come, go, and/or fly off the edge of the board, until it finally reaches a steady state after 1176 generations.

Allocating your arrays

There are two ways in *C* to create an array dynamically at run-time:–

- Use the **`malloc()`** function to allocate memory from *The Heap* and return a pointer to that memory. This is the most common practice in *C*. We will study it in class shortly.
- In **`gcc`** or *C++,* inside a function or compound statement, declare an array whose size is specified by an expression at run time. For example, the following is legal in **`gcc`**:–

```
void function(unsigned int x, …) {
   int A[x], B[x], C[x];

   /* use arrays A, B, and C */
   …
}  //   Function
```

Unfortunately, this only works for single-dimensional arrays. For a multi-dimensional array, only the first subscript can be determined at run-time. The rest of the subscripts must be compile-time constants.[2] Because this assignment calls for the grid of the Game of Life to be determined at run-time (in both dimensions), you cannot use it.

Allocating multi-dimensional arrays at run-time

§5.9 of Kernighan and Ritchie describe how the effect of a two-dimensional array can be achieved by allocating an array of pointers, each pointer of which points to another array of **`int`**. Suppose that you wish to allocate an array *B* of **`x`** rows of **`y`** columns each. One way is as follows:–

```
int *B[ ];
unsigned int i, j, k;

B = malloc(x * sizeof(int *));
if (B) for (i = 0; i < x; i++) {
   B[i] = malloc(y * sizeof (int));
   if (!B[i]) exit(-1);      /* error */
}
```

Then the array element of row **`j`**, column **`k`**, may be accessed as follows:–

```
B[j][k] /* assuming that j < x and k < y */
```

There are other ways of solving the same problem.

Playing the Game

To play the game, it is suggested that you set up a function along the lines of the following to play one generation:–

```
void PlayOne (unsigned int x, unsigned int y, int Old[ ][ ], int
New[ ][ ]) {
   /* loop through array New, setting each array element to zero
or
         one depending up its neighbors in Old.*/
}  // PlayOne
```

[2] Kernighan and Ritchie do not allow dynamically-sized arrays at all. However, they do allow arrays with an unspecified size to be passed as arguments to function. In the case of multi-dimensional arrays, only the first subscript may be unspecified; the remaining subscripts must be known at compile time. This is discussed on page 112.

This can be called by the function **Life** using:–

```
PlayOne(x, y, A, B);
```

The result is that **PlayOne** reads the contents of the first array (argument **A**) and updates the second array (argument **B**). Subsequent generations might be played by

```
PlayOne(x, y, B, C);
PlayOne(x, y, C, A);
```

so that the generations cycle among three arrays. If the **pause** argument is set to **'y'**, the program should wait for one character of input from the keyboard between calls to **PlayOne()**.

To test for termination conditions, you could adapt **PlayOne** to return values of zero or non-zero to indicate whether anything has changed. You should also construct another function to compare two arrays, returning zero if they are the same and non-zero if they are different, for example.

Testing

You should test your Game of Life with several initial conditions, including patterns that you find on the web. When the graders test your program, they will use one or more standard input files containing with typical patterns. The program arguments will match the input files.

Deliverables

This project must be carried out on the course virtual machine. Your submission must include the following:–

- *Two* or more **.c** files and one or more **.h** file to implement your game.
- A **makefile** to build your assignment. The executable program must be called **life**. The **makefile** must be able to make any individual **.o** file or the entire application. It also must be able to **make clean**.
- At least one test case that demonstrates that your program works on a non-trivial pattern.
- A document called **README.txt**, **README.pdf**, **README.doc**, or **README.docx** summarizing your program, how to run it, and detailing any problems that you had. Also, if you borrowed all or part of the algorithm for this assignment, be sure to cite your sources *and* explain in detail how it works.

Before submitting your assignment, execute **make clean** to get rid of extraneous files. Then export your project and your test case to an archive zip file named **PA1_userName.zip**, where **userName** is replaced by your WPI username (i. e., login ID). Submit that zip file, along with some output from your test cases and your **README** file, to *Canvas*.

This programming assignment is named **PA1**. Programs submitted after Saturday, January 21, 2017, 6:00 PM, will be tagged as late, and will be subject to the late homework policy.

Grading

This assignment is worth forty (40) points. *Your program* must *compile without errors in order to receive* any *credit*. It is suggested that before your submit your program, compile it again on a different platform

from the one you have been using, just to be sure that it does not blow up and does not contain surprising warnings.

- Correct **`makefile`** to build program and individual components and to clean up – 4 points
- Correct compilation without warnings (using **`-Wall`** switch – 4 points
- Correctly reading the initial configuration and centering it in the array – 4 points
- Correct allocation of arrays at run time – 4 points
- Correct use of two-dimensional arrays – 4 points
- Correct implementation of game function – 4 points
- Correct test for termination – 4 points
- Satisfactory test cases – 4 points
- Correct execution with graders' test cases – 4 points
- Satisfactory **`README`** file, including loop invariants – 4 points

Additional Notes

Command line arguments in *C* are explained in §5.10 of Kernighan and Ritchie. The function prototype of **`main()`** is

```
int main(int argc, char *argv[]);
```

The elements of the **`argv`** array are strings, which we have not yet studied in this course. The argument **`argv[0]`** contains the name of your program, **`argv[1]`** is the value **`X`**, **`argv[2]`** is the value **`Y`**, **`argv[3]`** is the value **`gens`**, and **`argv[4]`** is a string containing the name of the input file. The numeric values can be extracted using the function **`atoi()`**. The file name can be used directly in calls to **`fopen()`** . Sample usage is shown below:–

```
#include <stdio.h>
#include <stdlib.h>

FILE *input;
int k, x, y, gens;

if (argc < 5)
   /* report error in command line */

x = atoi(argv[1]);
y = atoi(argv[2]);
gens = atoi(argv[3]);

input = fopen(argv[4], "r");
if (!input)
   /* report unable to open file */

/* continue with print and pause arguments */
```

Project 3 (40 points)
Assigned: Monday, January 30, 2017
Due: Monday, February 6, 2017, 6:00 PM

CS-2303, System Programming Concepts, C-term 2017

Programming Assignment #1 — Binary Trees

Abstract

Write a program in *C* to scan one or more text files and count the number of occurrences of each word in those files. Use a binary tree to keep track of all of the words. When all input files have been scanned, print out a sorted list of the words to another file, along with the number of occurrences of each one. For extra credit, set up your tree to be an AVL tree, so that it is always balanced.

This project is similar to Assignment #HW5 in CS-1004, Introduction to Programming for Non-majors, in A-term 2016 as described here:– docx, pdf. In that assignment, students wrote in *Python* and used a *Python dictionary* as the principal data structure. This assignment is in *C* and requires a *binary tree* as the principal data structure.

Outcomes

After successfully completing this assignment, you should be able to:–

- Define a **`struct`** and organize your *C* program in an object-oriented style.
- Build a massive, recursive data structure comprising those objects.
- Search for an item in that data structure and, if it is not found, add it to the structure.
- Create a file for your output and write to it.
- Carry out simple string manipulation
- Put together a non-trivial program in *C*.

Before Starting

Review the following sections in Kernighan and Ritchie:–

- §5.1–5.2 regarding pointers;
- §§6.1–6.5 regarding **`structs`** and self-referential data structures;
- §7.5 regarding the creation, opening, and writing of files; and
- §7.8.5 regarding **`malloc()`** and **`free()`**.

This Assignment

Your program should be called **PA1**. Your program must accept an indeterminate number of arguments on the command line. The first argument specifies the output file, and the remaining arguments specify a sequence of input files. Thus, a user could invoke your program from a command line by typing

```
./PA1 outputFile inputFile1 inputFile2 ...
```

Under this command, the program would open and read each input file in turn, building up a *binary tree* of words and counts as it progresses. Once all files have been read and closed, the program must create the output file and write out the words in the tree *in alphabetical order*, one word per line, along with the number of occurrences of that word. Your program should ignore the case of the words, so that "**This**" and "**this**" are considered the same. However, words that are actually spelled differently — such as "**car**" and "**cars**" — are considered to be different words. You should recognize contractions such as "**won't**" and possessives such as "**Bob's**" as words. You should also treat hyphenated words such as "**hard-hearted**" as one word, not two.

A sample output might look like the following:–

```
   166    a
    25    and
    11    as
     3    command
    15    each
     2    file
     4    files
   109    in
     4    input
    98    it
     1    it's
    99    of
     3    open
     6    program
    18    read
   152    the
    41    this
     3    under
    30    would
-------------
    19    Distinct words
   790    Total words counted (including duplicates)
```

To allow for very long input files, the field width of the number of occurrences of each word should be at least six decimal digits. You must total and print the number of distinct words *and also* the total number of words counted (including duplicates).

For debugging, you may use the following text files:–

http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/Kennedy.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/Obama.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/MartinLutherKing.txt

You may also use other non-trivial documents (e.g., essays, newspaper or magazine articles, and speeches). Here are three Shakespeare plays:–

http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/Macbeth.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/MuchAdoAboutNothing.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/TamingOfTheShrew.txt

Finally, the following is a particularly long file containing the great American novel *Moby Dick*:–

http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/MobyDick.txt.

Implementation in C

You must implement this project in an object-oriented style. Think, for example, how you might have done it in Java, and then use that approach as a guideline for organizing your *C* program.

At the very minimum, you should define two or more **`.h`** files and a corresponding **`.c`** file for each **`.h`** file. Each **`.h`** file corresponds to a different part of your program — e.g., the tree management, input and output, etc. — much the same way that you would organize your classes in *Java*. In addition, you should also have a separate **`.c`** file for your **`main()`** function and its supporting functions.

For example, you might define a **`tree.h`** to be the interface to the binary tree. This would include the headers to functions for adding nodes to the tree, iterating through the tree, and deleting the tree (and all of its nodes). The companion **`tree.c`** would define the tree data structure (including the **`struct`** for the nodes the pointer to the root). It would contain the implementations of these functions. This would be the intellectual equivalent of a *BinaryTree* class in Java.

For input and output, you should use *C* functions such as **`fgets()`** and **`fprintf()`**.

Compiling C programs

To compile your *C* program, create a **`makefile`** along the lines of Laboratory #1. The compiler command line for **`.c`** files should be

```
gcc $(CFLAGS) -c yourFile.c
```

The make variable **`CFLAGS`** should default to **`-g0`**.

Note: Don't forget to delete your tree and free all of the nodes and strings in it before your program exits. Failure to do so can result in memory leaks. The graders will be looking for this.

String manipulation

A traditionally challenging part of this assignment is the handling of strings. Recall that a *string* in *C* is an array of characters terminated by the null character (i.e., **`'\0'`**). The recommended approach for this assignment is to do something similar to the *Python* version assigned to the CS-1004 class. That is:–

- read the text of each input document;
- partition it into substrings at "whitespace" boundaries;
- for each substring, "strip" the punctuation from the beginning and end;

- if the remaining substring is non-null, enter it into the binary tree data structure as a word.

"Whitespace" is defined at the space, newline, tab, vertical tab, form feed, and carriage return characters — i.e., any of the characters in the string **`" \n\t\v\r\f"`**. These define the boundaries between words and word-like substrings. An algorithm for splitting may be discussed in class.

Punctuation characters to strip from the beginnings and end of words typically include the characters **`".,?-;:()[]!\"\'"`**. If you find other characters that should be stripped in the input files, you may include them. An algorithm for stripping may be discussed in class.

Binary tree

Kernighan and Ritchie describe binary tree data structures in §6.5. You may base your implementation on the code in this chapter, but blindly copying that code will probably not meet the needs of this assignment. AVL trees (for extra credit) are *not* covered in Kernighan & Ritchie, but they were covered in CS-2102, Object-Oriented Programming.

Individual or Team Project

You may carry out this project individually, or you may optionally carry it out in a two-person team. In either case, you may consult classmates and others on the algorithms for creating and manipulating binary trees — for example by drawing pictures on a whiteboard or other temporary medium — but you must implement them in your program on your own or just with your teammate.

If you wish to work as a team, *you must register your team* with the professors so that it can be entered into *Canvas*. Your team name will consist of the WPI usernames of both members, in alphabetical order and separated by a hyphen. Once registered, either teammate may submit on behalf of the team.

Deliverables

You must carry out this project on the course virtual machine. You may use any tools, but you are strongly encouraged to use an IDE such as *Eclipse*. When you are ready to submit, clean the project and then export it to a *zip* file named **`PA1_userName.zip`** or **`PA3_teamname.zip`**, where **`userName`** is replaced by your WPI username and where teamname is the name registered in *Canvas*. The zip file should contain the following:–

- All of the *C* header and program files of your project, including your **`.c`** and **`.h`** files.
- A **`makefile`**. The target name should be **`PA1`**.
- Text files representing at least two non-trivial test cases other than the test cases cited above.
- The output of your test cases showing that the program works.

You must also submit a document called **`README.txt`**, **`README.pdf`**, **`README.doc`**, or **`README.docx`** summarizing your program, how to run it, and detailing any problems that you had. If you borrowed from or consulted with anyone on any part of the algorithm for this assignment, be sure to cite your sources. Your **`README`** file does not need to be part of the *zip* file.

Before submitting your assignment, execute **`make clean`** to get rid of extraneous files. The penalty for an "unclean" submission is 10 points. Submit your program to *Canvas* under the course named *CS2303-C17-LABS*. This assignment is named *PA1*. Programs submitted after the due date (Saturday, January 21, 2017, 6:00 PM) will be tagged as late, and will be subject to the late homework policy.

Grading

This assignment is worth forty (40) points. *Your program* must *compile without errors in order to receive* any *credit.*

- Correctly build from the **`makefile`** without warnings (with **`-Wall`** switch – 5 points
- Organization of program into at least three modules – 5 points
- Correct construction of binary tree and insertion of nodes – 5 points
- Correct traversal of binary tree and output of information according to specified format – 5 points
- Correct use of **`malloc()`** and correctly **`free`**ing all **`malloc`**'ed data – 5 points
- Proper destruction of tree and all of its objects before exiting – 5 points
- Correct execution with graders' test cases – 5 points
- Satisfactory **`README`** file, including output of two non-trivial test cases – 5 points

Note: A penalty of ten **points** will be assessed if your project is not *clean* before creating the zip file or for submitting your programs in some format other than a *zip* file.[3]

Note 2: If your program does not compile correctly on the course virtual machine using the `makefile`, the graders will attempt to contact you via e-mail. You will have 24 hours to resubmit a corrected version, and a penalty of 25% will be assessed (in addition to other penalties).

Extra Credit — AVL tree

For 25% extra credit, make your binary tree into an AVL tree, so that it is balanced at all times. In addition, submit a graph or a plot that shows the running times of the AVL tree algorithm *versus* the ordinary binary tree algorithm for at least three test cases including at least one small, one medium, and one large input. You should do this *after* you get the program working with an ordinary binary tree. Be sure to explain in your **`README`** file how your AVL tree works and how you know that it is balanced.

If you elect to do this extra credit, it is *strongly* suggested that you get the basic version of the project working first. Submit it as a zip file named **`PA1_userName.zip`**. Next, develop and debug the code to implement the AVL tree, and export that to a file named **`PA1_userName_AVL.zip`**. The graders will attempt to grade your AVL version first. If that does not work, they will fall back to the non-AVL version.

[3] The Professor's computer is **`tar`**-file challenged, **`rar`**-file challenged, as well as challenged for all other kinds of archive files.

Project 4 (45 points)
Assigned: Friday, February 3, 2017
Due: Monday, February 13, 2017, 6:00 PM

CS-2303, System Programming Concepts, C-term 2017

Programming Assignment #1 — Event Driven Simulation

Abstract

Write a *C++* program that simulates the activity of customers in queues at a bank.

Outcomes

After successfully completing this assignment, you should be able to:–

- Develop a non-trivial *C++* program
- Write a program that simulates some real-world activity
- Use linked lists in *C++*
- Use a random number generator

Before Starting

Read Chapter 6 of *Absolute C++*, which reviews **`structs`** from *C* and introduce classes in *C++*. Also read §7.1, which describes *constructors* and their use, and read §10.1 regarding **`new`**, **`delete`**, and *destructors*. Classes are, of course, at the heart of the *C++* language, and **`new`**, **`delete`**, constructors, and destructors are the principal means of creating and destroying objects of those classes.

Also read §17.1 of *Absolute C++*, which is a good introduction to the general concept of linked list.

Event-driven Simulations

An *event-driven simulation* is a computer program that mimics the behavior of people or objects in a system in response to events that occur at certain times. The program must maintain a data object for each person or object (called an *actor*) and place it in a queue according to the time of its event. It then reads the queue in the order of the events and, for each event, causes the corresponding actor to do its actions scheduled for that time. The action of the actor may be to change its own state, change the state of the system, do something on behalf of another actor, or something else.

Sometimes, the action will cause the actor to rejoin the event queue for a subsequent action. Sometimes, the action may add some other actor to the event queue or to another queue.

> *Example:* In a simulation of a radar system, an event might represent the transmission of a pulse at a certain time *t*. When simulated time *t* arrives, the simulator invokes the action (i.e., method) of the pulse event. This action enumerates the targets for the pulse and, for each target, computes the round trip time τ of the pulse and adds a new event to the event queue for time $t + \tau$ representing the return of the reflected signal.[4]

In this assignment, you will simulate customers arriving at a bank and standing in line in front of one of the tellers. People arrive at random intervals. Each person waits in his/her selected line until reaching the head of that line. When a person reaches the head of his line, the teller provides service for a random amount of time. After the service is completed, the person leaves the bank. The purpose of the simulation is to measure the average amount of time people spend between arriving at the bank and leaving the bank.

Assume that when there is a separate line for each teller, a newly arrived person joins the shortest line (or selects randomly among the set of equally short lines) and stays in that line until served. That is, no person leaves a line without being served, and no person hops from one line to another.

If a teller has finished serving a customer and there are no other customers waiting in its line, the teller selects the first customer from the line of another, randomly-chosen teller and serves that customer. If there are no customers waiting at all, the teller does other duties for a (small) random amount of time before checking the lines again.

The entire purpose of this simulation is to compare the performance of a single line serving all tellers *versus* separate lines for each teller.

Implementing your program

Your program should be called **`qSim`**. It needs to do several things:–

- Get and interpret the program parameters from the command line.
- Create a class object for each customer indicating his/her arrival time at the bank. Arrival times are determined from a uniform random number generator and the input parameters of the simulation. Also create a class object for each teller, with a random idle time in the range 1-600 seconds. *All constants in this simulation must be defined symbolically.*[5]
- Create a single *event queue* in the form of a linked list. The members of the linked list may be customers or tellers.
- Place each object in the *event queue* sorted according to the time of its event. That is, the event with the earliest time is at the head of the queue, and the event with the latest time is at the tail of the queue.
- Play out the simulation as follows:– take the first event off the *event queue*, advance a simulated *clock* to the time of that event, and invoke the action method associated with that event. Continue until the event queue is empty.

---

[4] This example is based on an MQP carried out in *C++* at MIT Lincoln Laboratory by WPI students in 2010. The simulator developed by these students is now in production use throughout Lincoln Laboratory and is used for testing the software of major defense radar systems.

[5] It is recommended that you use the preferred *C++* way of defining symbolic constants, i.e., as **`const`** declarations.

- Print out the statistics gathered by the simulation.

For this assignment, you will need to play the simulation twice — once for a bank with a single queue and multiple tellers and once for a bank with a separate queue for each teller. Draw some comparison about the average time required for a person to be served at the bank under each queue regime.

Here are some of the actions that can occur when an *event* reaches the head of the *event queue*:–

- If the event represents a newly arrived customer at the bank, add that person onto the end of a teller line — either the common line (in the case of a bank with a single line for all tellers) or to the shortest teller line. If there are several equally short teller lines, choose one at random.
- If the event represents a customer whose service at the bank has been completed, collect statistics about the customer:– in particular, how long has the customer been in the bank, from arrival time to completion of teller service. After collecting the statistics, the customer leaves the bank and its **`Event`** object is deleted.
- If the event represents a teller who has either completed serving a customer or has completed an idle time task, gather statistics about that teller. If there is no customer waiting in any line, put a teller event back into the *event queue* with a random idle time of 1-150 seconds.

  If there is a customer is waiting in line, remove the first customer from its line, generate a random service time according to the input parameters of the program, and add *two* events to the event queue, sorted by time. One is a customer event and represents the completion of that service. The other event is a teller event representing completion of a service and to look for the next customer (or to idle).

Class Hierarchy

You must define one or more classes that allow you to represent *Events*, *Customers*, and *Tellers*. It is suggested that the most important class of your simulation should be **`Event`**. How you distinguish between events associated with customers and events associated with tellers is your choice.[6]

Since there are two kinds of events — one representing tellers and one representing customers — **`Event`** itself should be an *Abstract Class* and **`Customer`** and **`Teller`** should be derived classes.

Two methods of an **`Event`** are to add it to the **`event queue`** and to remove it from the **`event queue`**. In addition, each **`Event`** has an **`Action`** method that is to be invoked when an **`Event`** is removed from the **`event queue`**. The **`Action`** method should be a *Pure Virtual* function, with concrete **`Action`** functions in each derived class. These should perform the appropriate action for a customer or teller, depending upon the type of the object that this action represents.

Each line in front of a teller should be implemented by an instance of a class called **`tellerQueue`**. For this assignment, do *not* attempt to use the **`queue`** container class from the Standard Library. Implement this class using a linked list, similarly to what you would have done in *C*. You will need to write methods to add customers to the end of the linked list and to remove them from the head of the list. In addition, include a *static variable* in the **`tellerQueue`** class that indicates which line

[6] An ideal representation would be to make **`Event`** and abstract class and to derive **`Customer`** and **`Teller`** from it. However, *abstract classes* are not scheduled to be introduced in this course until after the assignment is due. Therefore, you should choose a practical approach that enables you to complete the assignment on time.

(i.e., instance of the **tellerQueue** class) is the shortest. If more than one line is equally short, select one at random.

The **eventQueue** itself should also be implemented by a linked list. You will need to write a method to add an **Event** to the **eventQueue** in time order. This method should iterate through the list until it finds an **Event** with a time greater than the **Event** being inserted, and then it should insert the new **Event** just before that **Event**. You will also need a method to remove an **Event** from the **event queue** and invokes the **Action** method of the event. For extra credit (see below), you may implement a **priority_queue** class from the *Standard Template Library*.

Input

The command line of your program should be of the following form:–

```
./qSim #customers #tellers simulationTime averageServiceTime <seed>
```

The numbers of customers and tellers should be integers, and the simulation and average service times should be floating point numbers in units of minutes. The seed is optional and indicates a fixed seed for starting the random number generator (see below) For example,

```
./qSim 100 4 60 2.3
```

should be interpreted to mean that 100 customers and four tellers should be simulated over a period of 60 simulated minutes. The service time for each teller is an *average* of 2.3 minutes. No random number seed is specified.

Random Number Generation

To generate random numbers, use the function **rand()**, which is described in Figures 3.2 and 3.4 and the end of §3.1 of *Absolute C++*. These are also described on pages 46 and 252 of Kernighan & Ritchie. The function **rand()** is a *pseudo random number generator*. It generates a different number each time it is called, and those numbers look like they are random in the range **0 .. RAND_MAX**. In reality, however, it can generate the exact same sequence of "random" numbers repeatedly, to make it possible for you to debug your program. To use **rand()**, you need to include **<cstdlib>** and specify the appropriate **using** directive.

Random number generators work by maintaining one or more internal counters and performing a contorted transformation on the most recent number generated to get a new one that appears to be unrelated to the previous one. The random sequence can be initialized by calling **srand(seed)** at the beginning of your program,[7] where **seed** is an unsigned integer value. If you did not specify a **seed** in the command line, use some number that is likely to be truly random, such as the time returned from **time(NULL)**, which is also part of **<cstdlib>**. This seeds the random number generator to the current time.

To generate random arrival times with a uniform distribution, the following is suggested:–

```
float arrTime = simulationTime * rand()/float(RAND_MAX);
```
[8]

[7] Call **srand()** exactly once, preferably in the **main()** function after interpreting the command line arguments.

[8] The "function" **float(…)** is the constructor for an object of type **float** and converts the value of its argument into floating point number. It replaces the concept of *cast* from *C*.

It is useful to generate all customer arrivals at the beginning of the program and put them into the **event queue** in order of arrival time.

To generate random service times, the following is suggested:–

```
float serviceTime = 2*averageServiceTime*rand()/float(RAND_MAX);
```
[9]

Output

After your simulation has completed for both types of queuing regimes, you should print out a summary with the following information:–

- Total number of customers served and total time required to serve all customers
- Number of tellers and type of queuing (one per teller or common)
- Average (i.e., mean) amount of time a customer spent in the bank and the standard deviation
- Maximum wait time from the time a customer arrives to the time he/she is seen by a teller.
- Total amount of teller service time and total amount of teller idle time.

The information that you need to print should determine the statistics that you gather during the simulations.

Teams

You may optionally work in two-person teams. If you already were registered as a team in Programming Assignment #3, and if you wish to carry that team forward, you do not need to do anything to re-register for this assignment.

If you were not part of a team, or if you wish to change teammates, you need to "register" your team in *Canvas* by sending an e-mail to cs2303-staff@cs.wpi.edu. We assume that both teammates share the workload approximately equally, and both teammates will receive the same grade.

Deliverables

This project *must* be carried out on the course virtual machine using your favorite development environment. It must be named **PA4_username**, where **username** is replaced by your WPI login username. You must provide the following:–

- The exported project files of your project, including **.h** files and **.cpp** files to implement your simulation, and the **makefile**. The target of the makefile should be called **qSim**.
- At least three different test cases that show the behavior of the bank under both queuing regimes. Show the command line and the output.
- A document called **README.txt**, **README.pdf**, **README.doc**, or **README.docx** summarizing your program, how to run it, and detailing any problems that you had. It must explain how you represent *events*, *customers*, and *tellers*. Also, if you borrowed all or part of the algorithm for this assignment, be sure to cite your sources *and* explain in detail how it works.

[9] In a professional situation, you would probably generate service times with a Gaussian distribution. However, that is not necessary in this project, and it would be an added complication to an already difficult assignment.

- An analysis of your results — i.e., under what circumstances a single queue is better or worse than a queue per teller. You may include this analysis in your **`README`** document.

Before submitting your assignment, *clean* your project to get rid of extraneous files. Export your files as a single zip file from *Eclipse*, as explained at the end of *Lab 2*.

Submit to *Canvas*. This assignment is named *Programming Assignment* #1.

Grading

This assignment is worth forty-five (45) points. *Your program* must *compile on the course virtual machine without errors in order to receive* any *credit*. If you develop on a platform other the course virtual machine, please export it to the course virtual machine for testing, in order to avoid surprises.

- Correct compilation using **`make`** and **`g++ -Wall`** without warnings – 5 points
- Correct use of random number generator and seed – 5 points
- Satisfactory program organization into an understandable set of functions and modules (i.e., **`.cpp`** and **`.h`** files) and satisfactory use of symbolic constants – 5 points (subjective).
- Satisfactory class definitions for **`Customer`**, **`Teller`**, and **`Event`** – 5 points
- Satisfactory implementation of **`event queue`**, including insertion in time order – 5 points
- Satisfactory implementation and use of **`action`** methods for **`Tellers`** and **`Customers`** – 5 points
- Evidence of satisfactory testing, including output from test cases – 5 points
- Correct execution with graders' test cases – 5 points
- Satisfactory **`README`** file, including a discussion of the merits of common or per-teller queuing – 5 points

Extra Credit

For ten points of extra credit, define and implement the **`eventQueue`** class using the **`priority_queue`** class from the *Standard Template Library*. The priorities are the event times of the **`Event`** objects representing the customers and the tellers.

Note: A penalty of ten **points** will be assessed if your project is not *clean* before creating the zip file or for submitting your programs in some format other than a *zip* file.

Note 2: If your program does not compile correctly on the course virtual machine using the `makefile`, the graders will attempt to contact you via e-mail. You will have 24 hours from the time of the graders' email to resubmit a corrected version, and a penalty of 25% will be assessed (in addition to other penalties)

Project 5 (50 points)
Assigned: Monday, February 13, 2017
Due: Tuesday, February 21, 2017, 6:00 PM

CS-2303, System Programming Concepts, C-term 2017

Programming Assignment #1 — Operator Overloading

Abstract

Design and implement a class for rational numbers, and write a test program to exercise that class. Overload the usual arithmetic operators to apply to objects of this class. Also overload the stream insertion and extraction operators (**<<** and **>>**) to read in and print out rational numbers.

Outcomes

After successfully completing this assignment, you should be able to:–

- Design a class of numerical objects and provide arithmetic on them.
- Implement operator overloading.
- Implement **`friend`** functions.
- Understand reference parameters and reference results.
- Understand and deal with **`const`** in classes and parameters.

Before Starting

Read Chapter 8 of *Absolute C++*, which introduces operator overloading, friends, and references. This assignment is adapted from Programming Project 2 of that chapter.

Notice and Warning

There are many, many designs and implementations of rational classes available in print and on the web. *You may not refer to these.* You must design and develop your class *yourselves.* You may consult your classmates, the teaching assistants, and the professor. You may speak to others inside or outside the Computer Science department *only to the extent that they have not referred to any previous definition of a rational class* in *C++* or *Java*.

Optional Teams

You may, if you choose, work in a two-person team. To register your team for joint submission in *Canvas*, please send an e-mail to cs2303-staff@cs.wpi.edu. Teams should name their projects `PA5_teamName`, where `teamName` is the team name assigned by the instructor or course staff in *Canvas*. Existing teams from Programming Assignment #4 will automatically continue to this assignment. If you wish to change teams or break up a team, please let us know at the same address.

This Assignment

There are two parts to this assignment — the definition and implementation of the *Rational* class and the creation of a *test program* that tests and exhibits all of the features of that class.

The Rational class

A rational number is a number that can be represented as the quotient of two integers. For example, $\frac{1}{2}$, $\frac{3}{4}$, $\frac{64}{2}$, $\frac{32767}{65536}$, etc., are all rational numbers.[10] You can represent rational numbers as two values of type **`int`**, one for the numerator and one for the denominator. Your new class should be named **`Rational`**.

You need at least four constructors:–

- A constructor **`Rational(const int num, const int denom)`** to set the rational number to any legitimate value.
- A constructor **`Rational(const int wholeNumber)`** to set the numerator to the value of the argument and the denominator to *1*.
- A copy constructor **`Rational(const Rational &a)`**.
- A default constructor that sets the value of the **`Rational`** to zero (i.e., 0/1).

Overload the input and output operators **`<<`** and **`>>`** so that numbers can be input and/or output in the form **`1/2`**, **`15/32`**, **`300/401`**, etc. Note that the numerator and/or denominator may contain a minus sign, so input values of the form **`-1/2`**, **`15/-32`**, and **`-300/-401`** are legal.

Overload the following operators so that they correctly apply to the type **`Rational`**:–

`==`, **`!=`**, **`<`**, **`<=`**, **`>`**, **`>=`**, **`+`**, **`-`**, **`*`**, and **`/`**

In addition, define and implement a conversion function **`toDouble(const &Rational)`** that converts a **`Rational`** number to a **`double`**.[11]

Your class and all of its constructors and operators must *normalize* the rational number — that is, reduce it to its lowest terms. For this, you should use *Euclid's algorithm*, described here:–

http://en.wikipedia.org/wiki/Euclid_algorithm

[10] By the quotients in this sentence, we mean everyday fractions, not the integer division result in *C* or *C++* that this expression would produce.

[11] It would be desirable to overload **`operator=`** to assign to **`double`**, but that is not possible under the "Rule on Overloading Operations" at the end of §8.2 of *Absolute C++*.

This should be applied to any numerator-denominator pair that is not already known to be normalized. If the result is negative, the minus sign *should be in the numerator*. For example, 4/–8 would be normalized to –1/2.

Hints: Two rational numbers $\frac{a}{b}$ and $\frac{c}{d}$ are equal if *a*d* equals *b*c*. If *b* and *d* are positive, *a*/*b* is less than *c*/*d* provided *a*d* is less than *c*b*.

Simplifications: An input of a zero for a denominator is illegal. In normal circumstances, your class should "throw an error." If you are comfortable throwing errors, you may do so. However, since we have not yet covered error handling in this course, a simpler approach is to set both the numerator and denominator to zero and to print an error whenever such a number is used as an operand to any operator.

You may also ignore overflows. That is, multiplication and division may produce results so large that even when normalized, the numerator or denominator may not fit into variables of type **`int`**. However, your operators should hold intermediate values in variables of type **`long long int`** so that you do not gratuitously lose information before normalizing.

The test program

Your test program should be called **`PA5`**. It must accept an indeterminate number of arguments on the command line, each of which specifies an input file containing a sequence of lines. Each line represents a rational expression to be evaluated as described below. Your program should be invoked from a command line by typing

```
./PA5 inputFile1 inputFile2 ...
```

Under this command, the program would open and read each input file in turn. The file may contain multiple lines. Each line would be an expression of rational numbers and operations in *postfix* form. That is, both operands precede their operator. You must scan the line, convert numeric input into rational numbers, apply each operation to the two operands preceding it, and output the (normalized) result in both rational and double format. For example, the input line

```
1/3 1/6 +
```

Means $(\frac{1}{3} + \frac{1}{6})$ and would result in the output line

```
1/3 1/6 +     : 1/2 (double 0.5)
```

That is, the output line should repeat the input line, followed by a tab and a colon, and followed by the answer in rational and double format.

Likewise, an input line of the form

```
1/4 1/8 + 2 *
```

should output

```
1/4 1/8 + 2 *      : 3/4 (double 0.75)
```

Boolean operators should output **`true`** or **`false`** (with no double equivalent) — e.g.,

```
3/4 6/8 ==    : true
```

If you encounter an error in an input line, or if the line is unintelligible, print an error message and proceed to the next input line. When you open a new input file, print a line identifying the file name before scanning and interpreting any input.

Design sufficient test expressions to exercise the entire class and all of its operators. *You may share and exchange test input files with your classmates.* Report the results of your testing in your **`README`** file.

Deliverables

You should carry out this project in *Eclipse CDT*. When you are ready to submit, clean the project and then export it to a *zip* file. The zip file should contain the following:–

- All of the *C++* files of your project, including your **`.cpp`** and **`.h`** files of the **`Rational`** class and one or more **`.cpp`** and **`.h`** files for your test program.
- A **`makefile`**. The target name should be **`PA5`**.
- One or more test files containing lines of the form described above. Each test file name should end in **`.txt`** and should begin with your user ID (or, if a team, the team name as assigned in *Canvas*). If you have more than one test file, suffix the names with sequence numbers or letters or some other identifying information

You must also submit a document called **`README.txt`**, **`README.pdf`**, or **`README.doc`** summarizing your program, how to run it, and detailing any problems that you had. If you borrowed any part of the algorithm or any test case for this assignment, be sure to cite your sources. Your **`README`** file does *not* need to be part of the *zip* file.

Before submitting your assignment, execute *make clean* to get rid of extraneous files. Submit it to *Canvas* under the assignment *Programming Assignment #5* (PA5). Programs submitted after the due date and time specified in *Canvas* will be tagged as late and will be subject to the revised late assignment policy for this course.

Grading

This assignment is worth fifty (50) points. Your program must be accompanied by a **`makefile`**, and the **`makefile`** must *compile your program without errors in order to receive* any *credit.*

- The Rational class itself – 25 points, allocated as follows:–
  - Correct definition of class and methods, and overloaded operators – 4 points
  - Correct implementation of four different constructors, of one destructor, and an internal normalization method – 6 points
  - Correct implementation of 13 operators (i.e., **`<<`**, **`>>`**, **`!=`**, **`==`**, **`<`**, **`<=`**, **`>`**, **`>=`**, **`+`**, **`-`**, **`*`**, **`/`**, and conversion to **`double`**) – 13 points
  - Compile without warnings using **`-Wall`** – 2 points
- The Test program – 10 points
  - Correct processing of command lines, opening and closing of input files – 3 points
  - Correct scanning and parsing of input lines – 3 points
  - Correct evaluation of parsed input lines and output of results – 3 points
  - Compile without warnings using **`-Wall`** – 1 points

- Test cases – 10 points
  - Enough lines in test files to exercise each comparison operation at least 3 times with a range of input values – 3 points
  - Enough lines in test files to exercise each arithmetic operator (**+**, **–**, *****, **/**) at least three times with non-trivial denominators, including denominators that are relatively prime and those that are not relatively prime – 4 points
  - Enough test cases to show that normalization works in all situations – 3 points
- Satisfactory **README** file explaining your class and your testing and showing the output of all test cases – 5 points
- Penalty of ten points if your project is not clean before creating the zip file or for submitting your programs in something other than a zip file.

Note: This program must be developed in *Eclipse*. The graders will grade it on machines equivalent to the course virtual machine.

Extra Credit

For ten points of extra credit, define and implement the assignment operators **=**, **+=**, **-=**, ***=**, and **/=**. Also, construct test cases that show that they work. Then extend the input lines of the test cases to include simple variables that can be assigned to on one line and used on later lines of the same test case. (Variables do not carry across from one input test file to another.)

Make sure that the test cases for extra credit are clearly identified and separate from the regular test cases of this assignment.

For five additional points of extra credit, implement a **friend** function to overload the assignment operator to assign a rational number to a **double**. Extend the test program to demonstrate that this works. In your **README** document, explain how your tested it and how the graders can replicate you tests.

Project 6 (50 points)
Assigned: Tuesday, February 21, 2017
Due: Thursday, March 2, 2017, 6:00 PM

CS-2303, System Programming Concepts, A-term 2017

Programming Assignment #6 — Polymorphism

Abstract

Implement a simple 2D predator-prey simulation using derived classes and virtual functions.

Outcomes

After successfully completing this assignment, you should be able to:–

- Design an abstract base class and several derived classes from the base class.
- Design one or more virtual functions in the base class and provide concrete implementations of them in the derived classes.
- Enumerate the objects and invoke a method on each one.

Before Starting

Read Chapters 14 and 15 of *Absolute C++*, which introduce inheritance and polymorphism, respectively. This assignment is adapted from Programming Project 3 of Chapter 15. Depending upon your approach to this assignment, you may also find the following useful:– §7.3 about vectors and §17.3 about iterators.

Teams: You may optionally work in two-person teams. To register your team in *Canvas*, please send an e-mail to cs2303-staff@cs.wpi.edu. Your team should make one joint submission, and the names of both team members must be on each file.

Existing teams from Programming Assignment #5 will be carried over to this assignment. If you wish to dissolve or change teams, please send e-mail to the same address.

This Assignment

This program involves a simulation of a grid of *n-by-n* squares, some of which may be occupied by *organisms*. There are two kinds of *organisms* — *doodlebugs* (the predators) and *ants* (the prey). Only one

organism may occupy a cell at a time. Time is simulated in steps. Each organism attempts to perform some action every step. No action may cause an organism to move off the edges of the grid.

Ants behave as follows:–

- *Move.* For every step, each ant enumerates its adjacent cells — *up*, *down*, *left*, or *right* — and randomly selects an unoccupied one that is on the grid. If all adjacent cells are occupied or off the edges of the grid, the ant does not move but rather remains in its current location. [12]
- *Breed.* If an ant survives for at least three time steps, at the end of the third time step (i.e., after moving) the ant gives birth to a new ant in an adjacent cell (i.e., *up*, *down*, *left*, or *right*). If more than one empty cell is available, it chooses one at random. If no empty cell is available, no birth occurs.[13] Once an offspring is produced, an ant cannot produce another offspring until it has survived three additional steps.[14]

Doodlebugs behave as follows:–

- *Move.* For every time step, each doodlebug moves to an adjacent cell containing an ant and eats that ant. If more than one adjacent cell contains an ant, one is chosen at random. The ant that was eaten is removed from the grid. If no adjacent cell (i.e., up, down, left, or right) contains an ant, the doodlebug moves according to the same rules as ants. Note that a doodlebug cannot eat another doodlebug.
- *Starvation.* If a doodlebug has not eaten an ant within three time steps, at the end of the third time step, it dies of starvation and is removed from the grid.
- *Breed.* If a doodlebug survives for at least eight time steps, at the end of the eighth time step it spawns off a new doodlebug in the same manner as an ant. If no adjacent cell is empty, no breeding occurs. Once an offspring is produced, a doodlebug cannot produce another offspring until it has survived eight additional steps. Starvation takes precedence over breeding; that is, a starving doodlebug cannot breed.

During each time step, the doodlebugs act before the ants. That is, a doodlebug may eat an ant that was about to move and possibly to breed; as a result, that ant is dead and can no longer do either.

If an organism (i.e., an ant or a doodlebug) is eligible to breed but prevented from doing so by virtue of no empty adjacent cells, it remains eligible to breed on the next step.

This Assignment

Write a program to implement this simulation and draw the world using the ordinary characters **`'o'`** and **`'x'`** representing ants and doodlebugs, respectively. Create an abstract class called *Organism* that encapsulates basic data common to ants and doodlebugs. This class should have a virtual function called **`move()`** that is defined in the derived classes *Ant* and *Doodlebug*. You will also need a representation of the grid itself, and each cell of the grid should contain the *null* pointer (if empty) or a pointer to an **`Organism`**.

---

12 You will have to establish a systematic way of ordering the actions of the ants. Obviously, a previous ant could move into the only possible space available to another ant, thereby preventing the second one from moving.

13 This would be highly unusual, because the cell from which the ant moved would now be vacant.

14 Obviously, if the ant cannot move, it also cannot breed, because there is no empty adjacent cell into which could be occupied by the newly born ant.

Program Arguments

The command line to run your program should resemble the following, where each of the arguments is an integer, and where any of the arguments (plus the following ones) may be defaulted:–

```
./PA6 gridSize #doodlebugs #ants #time_steps seed pause
```

6. **`gridSize`** — an integer representing the number of cells along one dimension of the grid (defaulting to 20)
7. **`#doodlebugs`** — an integer indicating the number of doodlebugs (default 5)
8. **`#ants`** — an integer indicating the number of ants (default 100)
9. **`#time_steps`** — the number of time steps to play (default 1000)
10. **`seed`** — an integer indicating the seed for the random number generator, with zero meaning to use the current time as the seed (default 1)
11. **`pause`** — an indication as to whether to pause. Blank or zero means do not pause. A non-negative value *n* means pause and print the grid after each *nth* step. Wait for one character input before continuing.

You may represent your grid in any way that you choose. For example, it may be two-dimension array similar to the ones we created for the Game of Life, or it may be an array of pointers to vectors, or it may be some other two dimensional data structure that is easy to access by indexes in the *x* and *y* directions. Each element of the grid should be an **`Organism *`** — i.e., a pointer to an object of type **`Organism`**.

Before the start, the specified number of Ants and Doodlebugs should be placed on the board at random locations.

Termination

The simulation should terminate after the number of steps specified on the command line or when all of the ants or doodlebugs are gone. After termination, print to **`cout`** a summary of the simulation, including

- the original command line as represented by **`argv`**,
- the number of steps simulated,
- the total number of ants during the course of the simulation and the number remaining,[15]
- the total number of doodlebugs in the course of the simulation and the number remaining, and
- a picture of the final grid.

[15] By "total number" in these two bullets, we mean the number of ants or doodlebugs at the start of the simulation plus the number of successful births.

Deliverables

You should create this project as a "**makefile** project with existing code" in *Eclipse CDT* as described in Lab #2.[16] When you are ready to submit, *clean the project* and then *Export* it to a **zip** file, also as described in Lab #2. The zip file should be named **PA6_username** or **PA6_teamName**, where **username** is replaced by your WPI login identifier, and where **teamName** is the name of the team as assigned in *Canvas*. The zip file should contain the following:–

- All of the *C++* files of your project, including your **.cpp** and **.h** files of your base and derived classes, plus at least one **.cpp** file for your **main()** function and simulation control.
- A **makefile**. The target name should be **PA6**. The **makefile** must be such that the graders can use it to build your project *outside of Eclipse.*[17]
- The output of at least two different test cases.

You must also submit a document called **README.txt**, **README.pdf**, **README.doc**, or **README.docx** summarizing your program and its principal classes and methods, how to run it, and detailing any problems that you had. If you borrowed any part of the algorithm or any test case for this assignment, be sure to cite your sources. Your **README** file should not be part of the *zip* file.

Before submitting your assignment, execute **make clean** to get rid of extraneous files, including *Debug* directories. Submit to *Canvas* under this assignment, which is named *PA6 -- Polymorphism.* Programs submitted after 6:00 pm on due date (March 2, 2017) will be tagged as late. Since this is the end of the term, no there can be no forgiveness for late assignments.

Grading

This assignment is worth fifty (50) points. *Your program* must *compile without errors in order to receive* any *credit.*

- The abstract **organism** class – 5 points:–
  - Correct definition of class and virtual methods – 5 points
- The **ant** subclass – 9 points:–
  - Correct definition of subclass – 3 points
  - Correct implementation of **move()** function – 3 points
  - Correct implementation of breeding – 3 points
- The **doodlebug** subclass – 12 points:–
  - Correct definition of subclass – 3 points
  - Correct implementation of **move()** function – 3 points
  - Correct implementation of breeding – 3 points
  - Correct implementation of eating – 3 points

[16] The easiest way to do this is to replace the *Lab 2* files with your own and then edit the *Lab 2* **makefile** to refer to the names of the **.c** and **.h** files of this project.

[17] This happens automatically if you create the project and export it *exactly* as specified in *Lab 2*. However, it is worth checking to make sure.

- Simulation framework – 15 points:–
  - Grid implementation – 5 points
  - Primary simulation loop invoking the **`move()`** methods of organisms – 5 points
  - Processing command line arguments and setting up initial configuration – 3
  - Satisfactory output of steps of the simulation – 2 points
- Satisfactory **`README`** file explaining your class and your testing and showing the output of all test cases – 5 points
- Satisfactory execution of graders' test cases – 4 points
- Penalty of ten points if your project is not clean before creating the zip file or for submitting your programs in something other than a zip file.
- If the graders cannot build your program by executing **`make`** in a Linux command shell, your grade will be zero. *There will not be an opportunity to fix it before the end of the term.*

It is therefore in your interest to be doubly sure that your program compiles correctly.

The best way to do this is to build and run it exactly as the graders will — i.e., download the image as submitted to *Canvas,* unzip it to an empty directory *outside* your normal directory hierarchy, type the `make` command, and then run the program with a suitable command line, preferably one that you reported in your `README` file.