CS246—Assignment 3 (Spring 2016)

R. Hackman B. Lushman

Due Date 1: Friday, June 10, 5pm
Due Date 2: Monday, June 27, 5pm

Questions 1, 2a, 3a, and 4a are due on Due Date 1; the remainder of the assignment is due on Due Date 2.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Moreover, the only standard headers you may `#include` are `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, and `<utility>`. Additionally you may use `<cstdlib>` in the question 1 solely for the `std::rand()` calls. Marmoset will be programmed to **reject** submissions that violate these restrictions.
Note: Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. Moreover, each question asks you to submit a `Makefile` for building your program. For these reasons, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your zip file, so that your zip file does not contain any extra directory structure.
Note: Questions on this assignment will be hand-marked to ensure that you are writing high-quality code, and to ensure that your solutions employ the programming techniques mandated by each question.
Note: You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do. A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Instead, we will provide compiled executables, suitable for running on `linux.student.cs`, that you can use to check intended behaviour. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

1. Mastermind is a 2-player boardgame wherein one player the "Codemaster" prepares a code of four colours from a set of six, the "Codebreaker" then repeatedly tries to guess the code and receives feedback from the Codemaster. The feedback given by the Codemaster is in the form of white and black pegs where each white peg represents a correct colour in the wrong position and a black peg represents a correct colour in the correct position. For this question you will create a class `Mastermind` that acts as the Codemaster in a variant of this game. In this variant the passcode, instead of being of length 4 can be any length greater than 0. Your `Mastermind` should take in three integers in its constructor, the first parameter being the random seed, the second the length of the code, and the last the number of guesses the "Codebreaker" has to guess the code. You may assume the codeLength and guessLimit being passed to your constructor are greater than zero. Each character in the passcode in our game will be one of the characters in the set (A, B, C, D, E, F). Consider the following object definition for the `Mastermind` class.

```
struct Mastermind {
  int seed;
  int codeLength;
  int guessLimit;
  char* code;

  Mastermind(int seed, int codeLength, int guessCount);
  ~Mastermind();
  playGame();
};
```

You must implement the undefined constructor, destructor, and `playGame` method to the following specifications. You may add any data members or helper methods you want to the class, and indeed you will need to add some data member to the class to hold the passcode in any way you see fit. For this problem you will include `cstdlib` and use the `srand` and `rand` functions as per the specifications. The specifications for password generation are very important as if you do not follow them exactly you will fail the test cases.

- Upon construction you must call `srand` once on the seed passed to your constructor.
- Upon a call to `playGame` you must first set the password, with exactly `codeLength` calls to rand without any other rand calls in your program, setting the characters in the `passcode` starting with index 0 until every character has been set. Each character should be set based on the remainder of the rand call when divided by 6, a 0 representing A, 1 representing B, etc.
- Once the password is set `playGame` should print out the welcome message and read in from `stdin` until you've received `codeLength` valid characters.
- When reading in characters from the user you must accept upper and lowercase characters from the set (A,B,C,D,E,F) as valid characters and ignore any other input.
- After each guess you should print out the number of black and white pegs the user got correct in form "Xb, Yw" where X and Y are the number of black and white pegs respectively. Then if the game has not been won or lost you should print out the next guess message which includes the number of guesses left and read in another guess.
- If the code is guessed before the guess limit is reached you should print the win message "You won in `guessCount` guesses!"
- If the code is not guessed before the guess limit is reached you should print the loss message "You lost! The password was:" and print the password on the following line.
- The number of black pegs given for each guess is equal to the number of characters the user has in the correct position in the code.
- The number of white pegs given for each guess is equal to the number of characters the user has that are in the code but in the wrong position, however the number of black and white pegs can never exceed the size of the code. This means that if a code has only one A and the user guesses 2 A's in the wrong position they receive only one white peg for A, additionally if the code has only one A and the user has guessed 3 A's, two in the wrong position and one in the correct position the user only receives the single black peg for the A's - no white peg.
- Once the game is over, win or lose, you must ask the user if they would like to play again and if they enter Y or y you must recursively call `playGame` otherwise you are done.

For example consider the following `playGame` call example to a Mastermind object that was initialized with 5, 4, 3. Lines preceeded with > represent user input. You must follow the exact message form as shown below.

```
Welcome to Mastermind! Please enter your first guess.
>AAAA
2b, 0w
2 guesses left. Enter guess:
>BCDE
1b,0w
1 guesses left. Enter guess:
>bcaa
3b,0w
You lost! The password was
BFAA
Would you like to play again? (Y/N): N
```

You will be provided a simple test harness `main.cc` that will take in four command line arguments corresponding to seed, codeLength, and guessLimit and constructs a Mastermind object with those values and calls playGame, do not modify this file.

Due on Due Date 1: Full implementation of the Mastermind class in C++. Your zip should contain, at minimum, the files `main.cc`, `mastermind.h`, `mastermind.cc`, and a `Makefile`. Additional classes must each reside in their own `.h` and `.cc` files.

2. The standard Unix tool `make` is used to automate the separate compilation process. When you ask `make` to build a program, it only rebuilds those parts of the program whose source has changed, and any parts of the program that depend on those parts etc. In order to accomplish this, we tell `make` (via a `Makefile`) which parts of the program depend on which other parts of the program. Make then uses the Unix "last modified" timestamps of the files to decide when a file is older than a file it depends on, and thus needs to be rebuilt. In this problem, you will simulate the dependency-tracking functionality of `make`. We provide a test harness (`main.cc`) that accepts the following commands:

   - `target: source` — indicates that the file called `target` depends on the file called `source`
   - `touch file` — indicates that the file called `file` has been updated. Your program will respond with

     `file updated at time n`

     where `n` is a number whose significance is explained below
   - `make file` — indicates that the file called `file` should be rebuilt from the files it depends on. Your program will respond with the names of all targets that must be rebuilt in order to rebuild `file`.

   A target should be rebuilt whenever any target it depends on is newer than the target itself. In order to track ages of files, you will maintain a virtual "clock" (just an `int`) that "ticks" every time you issue the `touch` command (successful or not). When a target is rebuilt, its last-modified time should be set to the current clock time. Every target starts with a last-modified time of 0. For example:

```
a: b
touch b
touch b
touch b
```

   will produce the output (on stdout)

```
b updated at time 1
b updated at time 2
b updated at time 3
```

It is not valid to directly update a target that depends on other targets. If you do, your program should issue an error message on stdout, as illustrated below:

```
a: b
touch a
```

(Output:)

```
Cannot update non-leaf object
```

When you issue the `make file` (build) command, the program should rebuild any files within the dependency graph of `file` that are older than the files they depend on. For example:

```
a: b
a: c
b: d
c: e
touch e
make a
```

will produce the output

```
e updated at time 1
Building c
Building a
```

because file `c` depends on `e`, and `a` depends on `c`. Note that `b` is not rebuilt. The order in which the `Building` messages appear is not important.

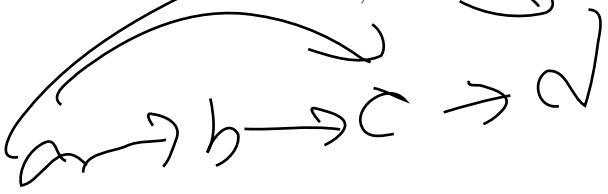
A file may depend on at most 10 other files. If you attempt to give a file an 11th dependency, issue the error message

```
Max dependencies exceeded
```

on stdout, but do not end the program. If you give a file the same dependency more than once, this does not count as a new dependency, i.e., if you give a file a dependency that it already has, the request is ignored. For example, if `a` depends on `b`, then adding `b` as a dependency to `a` a second time has no effect. On the other hand, if `b` also depends on `c`, then `c` may still be added to `a` as an additional dependency, even though `a` already indirectly depends on `c`.

There may be at most 20 files in the system. (Note that files are created automatically when you issue the `:` command, but if a file with the same name already exists, you use the existing file, rather than create a new one.) If you create more than 20 files, issue the error message

```
Max targets exceeded
```



on stdout, but do not end the program.

You may assume: that the dependency graph among the files will not contain any cycles. You may also assume that all `:` commmands appear before all `touch` commands, so that you do not have to worry about updating a leaf that then becomes a non-leaf because you gave it a dependency.

You may not assume: that the program has only one makefile, even though the provided test harness only manipulates a single `Makefile` object.

Implementation notes

- We will provide skeleton classes in `.h` files that will help you to structure your solution. You may add fields and methods to these, as you deem necessary.
- Do not modify the provided test harness, `main.cc`.
- For your own testing, because the order in which the `Building` messages occurs may be hard to predict, you may wish to modify your `runSuite` script to sort the outputs before comparing them. This does not provide perfect certainty of correctness, but it is probably close enough.

Deliverables

(a) **Due on Due Date 1**: Design a test suite for this program (call the suite file `suiteq2.txt` and zip the suite into `a3q2a.zip`)

(b) **Due on Due Date 2**: Complete the program. Put all of your `.h` and `.cc` files, and your `Makefile`, into `a3q2b.zip`).

3. Consider the following object definition for a mathematical set type for integers and a print method:

```
struct intSet {
  int *data;
  int size;
  int capacity;

  intSet();
  intSet(const intSet&);
  ~intSet();
  intSet(intSet &&is);
  intSet &operator=(const intSet &is);
  intSet &operator=(intSet &&is);
  intSet operator|(const intSet &other);  // Set union.
  intSet operator&(const intSet &other);  // Set intersection.
  intSet operator==(const intSet &other);  // Set equality.
  bool isSubset(intSet s);  // True if s is a subset of this set.
  bool contains(int e);  // True if e is an element of this set.
  void add(int e) // Adds int e to this set.
  void remove(int e); // Removes int e from this set.
};
std::ostream& operator<<(std::ostream& out, const intSet& is);
```

You are to implement the undefined constructors and destructors for the `intSet` type. Additionally you are to overload the bitwise or, bitwise and, equality, input, and output operators to represent set union, set intersection and set equality, reading in a set, and printing a set respectively. You must also implement the methods `isSubset`, `contains`, `add`, and `remove` above your methods and data must follow the rules below.

- Copy constructor and assignment operator must perform a deep copy so that each set has their own independant set of data.
- Each item in a set is unique, if a call to add passes an integer already in the set nothing is done.
- Set union returns a new set with all elements from both sets.
- Set intersection returns a new set with all elements that are in both sets.
- Two sets A and B are equal iff no element of one is not an element of the other. More precisely $(\nexists\, x\ s.t.\ x \in A \wedge x \notin B) \wedge (\nexists\, x\ st.\ x \in B \wedge x \notin A)$.
- `isSubset(intSet s)` returns true if every element in s is an element of the set the method has been called on, false otherwise.
- `contains(int e)` returns true if the integer e is an element of the set the method has been called on, false otherwise.
- After `add(int e)` is called the integer e must be in the set the method has been called on.
- After `remove(int e)` is called the the integer e must not be in the set the method has been called on.
- The capacity of the set should be initialized to 0 until the first integer is added at which point the capacity should be made to be 5. If at any point the capacity is not enough it should be doubled.
- When reading in a set you should read **one line** of whitespace-delimited integers from stdin adding each to the set.
- When printing a set you should first print a left parenthesis then print each integer in the set delimited by a comma and a space in ascending order and end with a right parenthesis. For example the set containing 3, 5, and 2 would be printed as follows: `(2, 3, 5)`

You will be provided with a test harness for this question. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use).

(a) **Due on Due Date 1:** Design a test suite for this program (call the suite file suiteq3.txt and zip the suite into a3q3a.zip).

(b) **Due on Due Date 2:** Full implementation of the intSet class in C++. Your zip should contain, at minimum, the files `main.cc`, `intSet.h`, `intSet.cc`, and a `Makefile`. Additional classes must each reside in their own `.h` and `.cc` files.

4. For this problem, the classes that make up the program are implemented using the `class` keyword. In this problem you will apply the Iterator pattern to create an iterator over binary search trees. A test harness `main.cc` is provided. Do not modify this file. When you have successfully completed this program, the "print" command `p` will display the contents of the binary search tree `t` in sorted order. Note that the provided `Tree` class stores information in each node about whether the node is a left child, a right child, or the root. Use this information to help you navigate the tree. Iteration must run in time $\Theta(L(x))$ where $L(x)$ is the length of the shortest path in the tree from x to x's successor node. In particular, you must perform the minimal amount of traversal in moving from a node to its successor.

   You may assume that all input is valid.

Due on Due Date 1: Design a test suite for this program (call the suite file `suiteq4.txt` and zip the suite into `a3q4a.zip`).

Due on Due Date 2: Full implementation in C++. Your zip file should contain, at minimum, the files `main.cc`, `tree.h`, `tree.cc`, and a `Makefile`. Additional classes must each reside in their own `.h` and `.cc` files.