

# CS246—Assignment 3 (Spring 2018)

C. Kierstead      A. Moss      V. Sakhnini

Due Date 1: Monday, June 11, 11:55pm

Due Date 2: Monday, June 18, 11:55pm

Due Date 3: Monday, June 25, 11:55pm

**Questions 1a, 2a, 3a, and 4a are due on Due Date 1; Questions 1b, and 2b are due on Due Date 2; Question 3b, and 4b are due on Due Date 3.**

**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>`, `<cassert>`, `<stdexcept>`, and `<utility>`. 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. Please review the CS246 style guidelines here:

<https://www.student.cs.uwaterloo.ca/~cs246/current/AssignmentGuidelines.shtml>

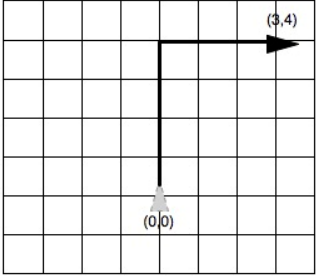
**Note:** You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do. A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential 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.

***Note: the material for due dates 1 and 2 are good review for the midterm!***

1. In this exercise, you will write a C++ class (implemented as a `struct`) to control a simple robotic drone exploring some terrain. Your drone starts at coordinates (0,0), facing north. Use the following structure definition for coordinates:

```
struct Position {
    int ew, ns;
    Position( int ew = 0, int ns = 0 );
};
```

The east-west direction is the first component of a position, and the north-south direction is the second. Your `Drone` class must be properly initialized via a constructor, and must provide the following methods:

<i>Method</i>	<i>Description</i>
<code>void forward()</code>	Move the drone one unit forward.
<code>void backward()</code>	Moves the drone one unit backward.
<code>void left()</code>	Turns the drone 90 degrees to the left, while remaining in the same location.
<code>void right()</code>	Turns the drone 90 degrees to the right, while remaining in the same location.
<code>Position current()</code>	Returns the current position of the drone.
<code>int totalDistance()</code>	Returns the total units of distance travelled by the drone.
<code>int manhattanDistance()</code>	<p>Returns the "Manhattan distance" between the current position and the origin where the Manhattan distance defined as the absolute north-south displacement plus the absolute east-west displacement.</p> <p>Note: If the drone starts at (0,0) facing north, and moves to position (3,4) where 3 is east-west and 4 is north-south, you've effectively moved 3 units east and 4 north. Therefore, the total would be 7 and not 5.</p> 
<code>bool repeated()</code>	Returns true if the current position is one that the drone has previously visited.

For simplicity, you may assume that the drone will never visit more than 50 positions before running out of fuel or otherwise breaking down.

Implement the specified operations for the `Drone`. (Some starter code has been provided for you in the file `drone.h`, along with a sample executable.) **You may not change the contents of `drone.h` other than by adding your instance variables and comments i.e. the interface must stay exactly the same.**

The test harness `main.cc` is provided with which you may interact with your drone for testing purposes. **The test harness is not robust and you are not to devise tests for it, just for the `Drone` class. Do not change this file.**

- (a) **Due on Due Date 1:** Design the test suite `suiteq1.txt` for this program and zip the suite into `a3q1a.zip`.
- (b) **Due on Due Date 2:** Implement this in C++ and place the files `Makefile`, `main.cc`, `drone.h` and `drone.cc` in the zip file, `a3q1b.zip`. Your `Makefile` must create an executable named `a3q1`. Note that the executable name is case-sensitive.

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 `cout`)

```
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 `cout`, 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 `cout`, 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 `cout`, but do not end the program.

**Note:** the file names can be any length i.e. any string is a valid file-name, they're just shown as single characters for ease of typing

**You may assume:** that the dependency graph among the files will not contain any cycles. You may also assume that all `:` commands 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 (should only modify a local copy of `runSuite` to sort the outputs, rather than your "master" `runSuite`) 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`. Your `Makefile` must create an executable named `a3q2`. Note that the executable name is case-sensitive.

3. In this question, you will implement a class that represents a mathematical set for integers (**without any duplicates**). The interface has already been implemented for you in `intSet.h`, make sure to read it carefully. Do not change the provided `intSet.h` **other than by adding your instance variables and comments i.e. the interface must stay exactly the same.**

Implement the interface in the implementation file `intSet.cc`. Your implementation must satisfy the following requirements:

- The copy constructor and the copy assignment operator must perform a deep copy so that each set maintains its data independently
- A set cannot contain duplicate items. Attempting to add an item already in the set does not change the set nor does it produce an error
- Set union returns a new set containing all the elements from both sets.
- Set intersection returns a new set containing only the elements that are in both sets.
- Two sets  $A$  and  $B$  are equal if and only if no element of one is not an element of the other. More precisely  $(\nexists x \text{ s.t. } x \in A \wedge x \notin B) \wedge (\nexists x \text{ s.t. } x \in B \wedge x \notin A)$  .
- `isSubset(const 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 `e` is an element of the set the method has been called on, false otherwise.
- After `add(int e)` is called, `e` must be in the set the method has been called on.
- After `remove(int e)` is called, `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 is increased to 5. If at any point the capacity is not enough, it should be doubled.
- The output operator first prints a left parenthesis, then each integer in the set delimited by a comma and a space in ascending order and ends with a right parenthesis. For example the set containing 3, 5, and 2 is printed as: `(2, 3, 5)`

A test harness is provided in `main.cc`. **Make sure you read and understand this code, 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 3 (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 3:** Full implementation of the `intSet` class in C++. Your zip file, `a3q3b.zip`, should contain at minimum, the files `main.cc`, `intSet.h`, `intSet.cc`, and a `Makefile` that creates the executable `a3q3`. Additional classes must each reside in their own `.h` and `.cc` files.

4. In this question you will implement a class that represents a date. The interface has already been implemented for you in `date.h` including partial documentations. **Make sure to read them carefully.** Do not change the provided `date.h` **other than by adding your instance variables and comments i.e. the interface must stay exactly the same.**

Implement the interface in the implementation file `date.cc`. Your implementation must satisfy the following requirements:

- The four setters should throw `out_of_range` exception (with an informative error message) if the consumed data is out of range (check comments for `Date` definition in `.h` file). Make sure to include `<stdexcept>`.
- The last six methods in `Date` class should return a `Date` in the valid range otherwise they should throw `out_of_range` exception

A test harness is provided in `main.cc`. **Make sure you read and understand this code, as you will need to know what it does in order to structure your test suite. Please note that the error message might be different than your choice.**

Note that we may use a different test harness to evaluate the code you submit on Due Date 3 (if your functions work properly, it should not matter what test harness we use).

- (c) **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`).
- (d) **Due on Due Date 3:** Full implementation of the `Date` class in C++. Your zip file, `a3q4b.zip`, should contain at minimum, the files `main.cc`, `date.h`, `date.cc`, and a `Makefile` that creates the executable `a3q4`. Additional classes must each reside in their own `.h` and `.cc` files.