ECE297 Milestone 1
Using and Creating Efficient APIs

"There is more to life than increasing its speed."

–Mahatma Gandhi

Assigned on Monday, Jan. 23

Due on Saturday, Feb. 11

TA: project management, code style, and git = 3/10

Autotester = 7/10

Total marks = 10/100

1 Objectives

This milestone focuses on using and extending an application programming interface (API). We have written a library called libstreetsdatabase that allows you to query geographic information database files for cities. This API consists of two pieces, each provided by a different header file. StreetsDatabaseAPI.h provides a *higher level* interface to structured street and intersection data; you will mostly use functions from this header file. OSMDatabaseAPI.h allows you to query *lower level* OpenStreetMap data on individual geographic points; you will only need to use this API for one function in this milestone.

In this milestone you will learn to use the API defined by these header files, and you will implement functions that will be useful for your project; essentially you are creating a new, richer API. To make some of the functions in your new API fast, you will also need to create and load some data structures of your own to allow efficient look-ups. You should make use of the STL container classes such as vectors, maps and/or unordered maps that you recently learned about in the course tutorials to build your data structures quickly. Some of the tests we have released for this milestone test the speed of your new API; based on the results of these tests you can see where you need to optimize for speed.

After completing this milestone you should be able to:

1	Query the libstreetsdatabase API using functions in the provided header files.
2	Create a new API of functions that will be useful in your project.
3	Use STL data structures such as vectors and maps, and choose appropriate data structures to speed up an API.
4	Use unit tests to test your code.

2 Problem Statement

In this milestone you will load a skeleton C++ project in your IDE and put it under revision control. This will be the C++ project on which you implement all of your remaining ECE297 milestones. You will then start using some of the functions (such as *getStreetName*) in StreetsDatabaseAPI.h and OSMDatabaseAPI.h which are located in `/cad2/ece297s/public/include/streetsdatabase`. These APIs allow you to access the data loaded from large binary files that describe all the streets and intersections in a city and more. You will then be asked to implement your own functions that use this API to provide higher-level functions. For example, you will implement a function that returns the names of the streets that meet at a given intersection. You will test your code using the **ece297exercise** autotester, and submit it using the **ece297submit** script. Note that you will be using **git** throughout the milestone to work effectively with your teammates.

3 Walkthrough

Even though your ECE297 project is divided up into milestones, all of the milestones are part of one project. You will be creating a mapping application similar to Google/Bing Maps. You will build upon the code you write in milestone 1 as you implement milestone 2, and so on.

! Your project is divided into milestones, but you will use your solutions to the milestones in all subsequent milestones.

This is also why you started off by learning how to use git. To build up your project code efficiently, it is important to keep it under revision control and to **commit** + **push** often. This will allow you to divide tasks easily among your team members, keep track of all changes made to your project code, and share the latest working code with your teammates. Note that your grade will depend not only on how well your code works, but also on your TA's assessment of your code style, usage of git, and project management.

1. Code correctness and performance will be automatically graded.
2. Project management will graded by your TA. Have you broken up the milestone into tasks assigned to each team member, and are the task owners, due dates and status tracked on your wiki page?
3. Your TA will also consider how well you use git: are there frequent commits by different team members, with good commit messages?
4. Coding style and commenting will be graded by your TA.

An effective team project is well-commented and uses revision control (such as git) to effectively divide work and maintain a history of code changes.

3.1 Project Setup

To setup the project *each team member* needs to run `ece297init 1` as shown below:

```
#The following command will setup the project and git repo
> ece297init 1
#Output trimmed...

You can now open your mapper project in your chosen IDE at:
  /homes/v/vaughn/ece297/work/mapper
#NOTE: the above path will differ based on your username.
```

Listing 1: Setting up the project

`ece297init 1` will create a git remote repository for all group members under `/groups/ECE297S/cd-XXX/mapper_repo` (where *XXX* is your group number), and commits the initial project files. It then clones a local repository and a working copy under `~/ece297/work/mapper`, which you can open as a Netbeans, Visual Studio Code, CLion or Eclipse project. You will use this project and repository for all the following milestones.

3.2 Code Organization

In this milestone you are building a library of higher level (more complex) functions that will be useful in later milestones. To build these higher level functions, you will call lower-level functions we have written for you that provide basic data from the OpenStreetMap database. The organization of these application programming interface (API) layers and of the code is shown in Fig. 1 and Fig. 2, respectively, and is detailed below.

- **libstreetmap/src**: This is the library that you will be creating throughout your ECE297 project. It contains “m1.cpp", which includes “m1.h". m1.h defines the interfaces of all the functions you will implement in milestone 1; you cannot change these interfaces and accordingly m1.h is a read-only file (in `/cad2/ece297s/public/include/milestones`) that you can read but can’t modify. You must write the implementation of all the m1.h function interfaces in files in libstreetmap/src; m1.cpp is a possible implementation file but you can create additional or different .cpp and .h files if you prefer. The comments in m1.h also give information about corner cases you should handle and how fast the various functions need to be; the `ece297exercise` autotester provides the precise speed specification by automatically testing the speed of these functions.
- **libstreetsdatabase**: This library provides the functions for accessing the libstreetsdatabase API that parses and interprets OSM data; it has been written for you. The most important header file is `StreetsDatabaseAPI.h`, and `OSMDatabaseAPI.h` is the next most important. These files and the other headers for this library are in /cad2/ece297s/public/include/streetsdataba

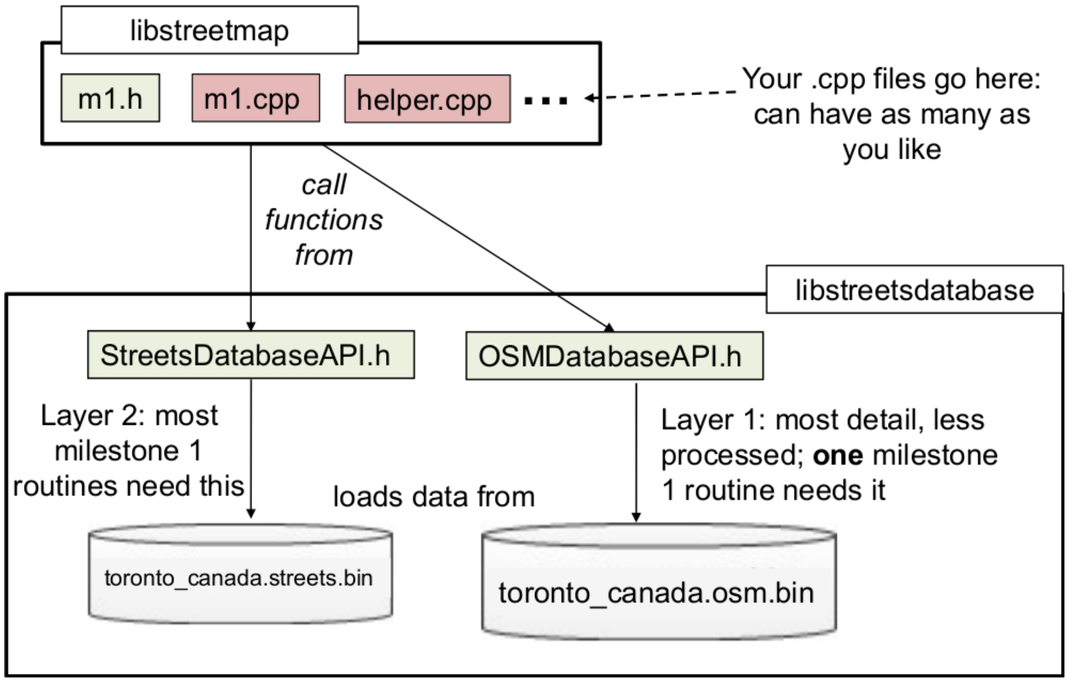

libstreetmap
m1.h
m1.cpp
helper.cpp
Your .cpp files go here: can have as many as you like
call functions from
libstreetsdatabase
StreetsDatabaseAPI.h
OSMDatabaseAPI.h
Layer 2: most milestone 1 routines need this
loads data from
Layer 1: most detail, less processed; **one** milestone 1 routine needs it
toronto_canada.streets.bin
toronto_canada.osm.bin

Figure 1: Project API layers.

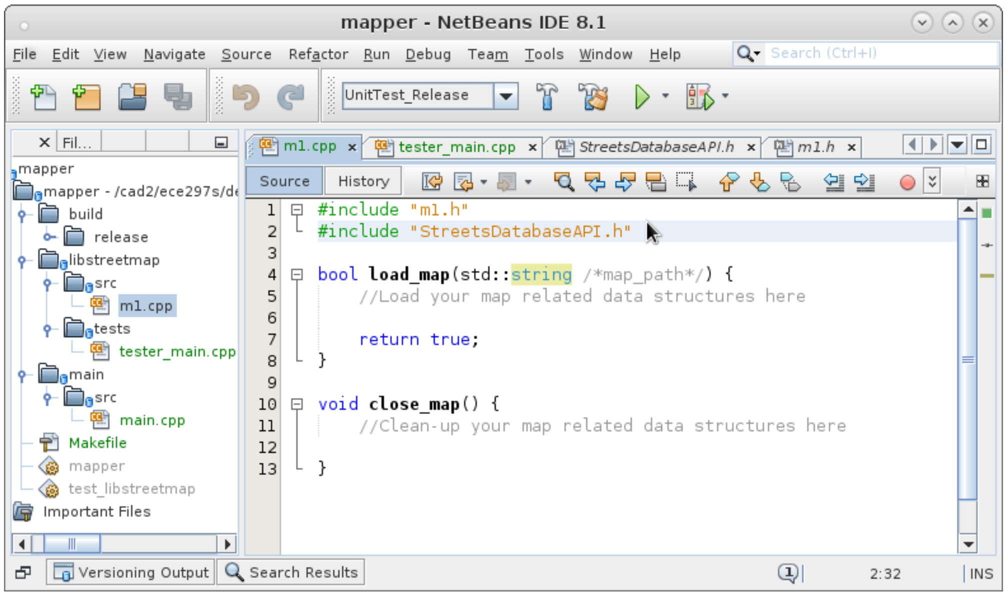
– you can read and use them but cannot modify them. An easy way to read these headers is to use NetBeans or another IDE to navigate into them.

StreetsDatabaseAPI.h is made up of several functions, and each function gives you information about streets or intersections that exist in a *cityname*.streets.bin file. Data for OSMDatabaseAPI.h is stored in a separate file called *cityname*.osm.bin. We'll be testing your code with the data files for several cities, including Toronto, New York, and Kyiv.

- **main**: This folder contains your "main.cpp" file; this is where your program starts executing when you type `mapper` at the command line. You can call the functions you wrote in libstreetmap from here. We have provided a simple main.cpp file that will simply load and close a map; you don't need to change it for milestone 1.
- **libstreetmap/tests**: Most of your testing in this course won't be performed with the `mapper` executable program that an end user would run. Instead, you will use *Unit Tests* that directly link to and test your libstreetmap functions. We have written unit tests for all the functions of this milestone; if you wish you can write additional tests and put the code for them in this folder.

3.3 Understand StreetsDatabaseAPI.h

Fig. 3 shows how a map is represented internally in the libstreetsdatabase. Each Intersection is a **graph node** and each Street Segment is a graph **edge** (which connects two Intersection nodes). Note that multiple Street Segments make up one Street – for example, the dotted Street Segments are all part of one Street (College St.).



```
#include "m1.h"
#include "StreetsDatabaseAPI.h"

bool load_map(std::string /*map_path*/) {
    //Load your map related data structures here

    return true;
}

void close_map() {
    //Clean-up your map related data structures here

}
```

Figure 2: Source code folders in NetBeans.

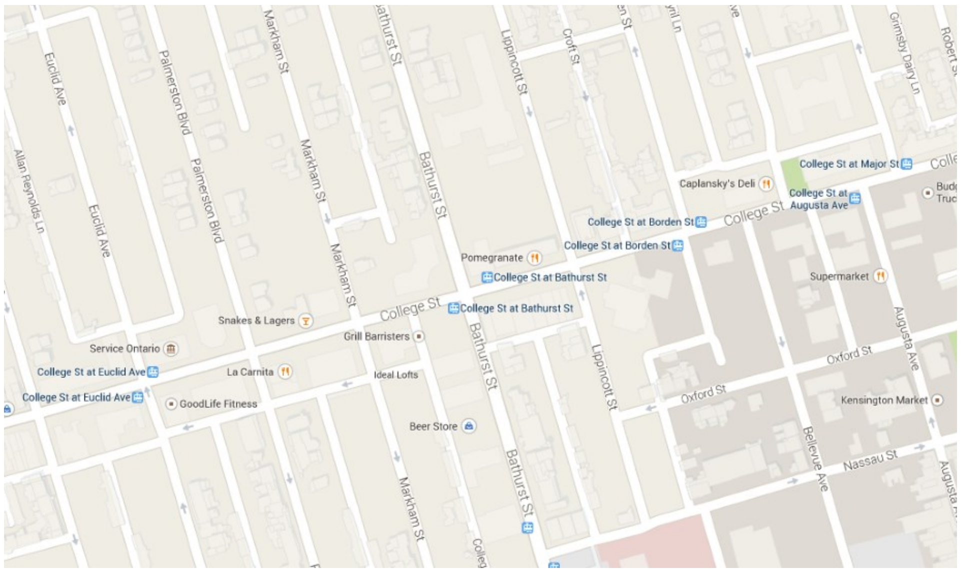
In the libstreetsdatabase graph, each graph node is an Intersection, and each graph edge (between 2 nodes) is a Street Segment.

The provided StreetsDatabaseAPI has an integer id for each of the intersections; from 0 to `getNumIntersections()-1` – which is the total number of intersections. Similarly, each Street Segment has an integer id from 0 to `getNumStreetSegments()-1` and each Street has an integer id from 0 to `getNumStreets()-1`. Additionally, each Intersection has a name, and each Street has a name. Note that it is possible for two Streets to have the same name; there is more than one *Main Street* in Greater Toronto for example. Intersection names are formed from the names of the streets that meet at that intersection. Street Segments do not have unique names associated with them.

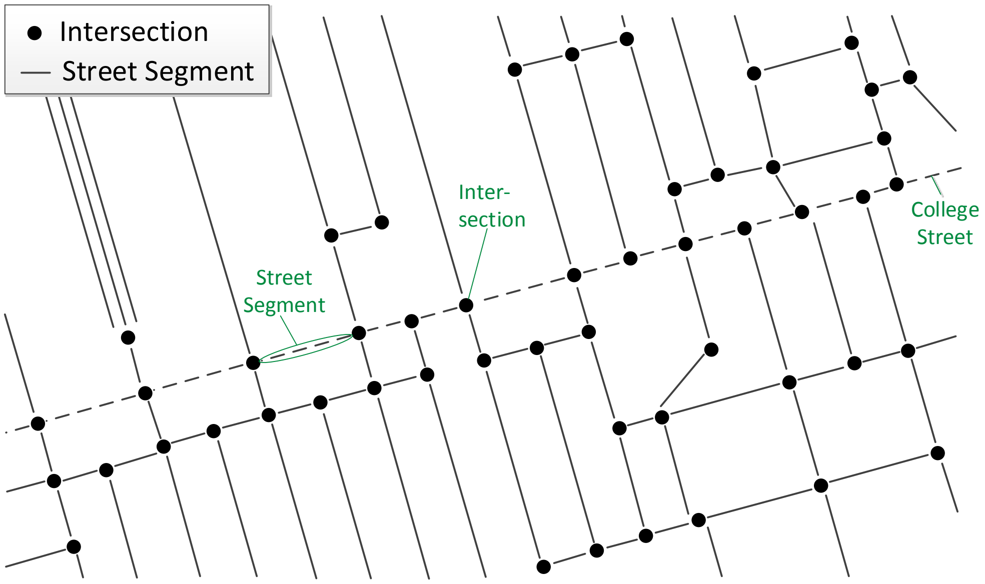
Each Intersection, Street Segment, or Street has a unique integer ID. Only Intersections and Streets have names, and they may not be unique.

Listing 2 shows "StreetsDatabaseAPI.h" which contains the libstreetsdatabase API functions that you can use to retrieve information about the map-graph of Toronto. All function prototypes are commented; this is the best kind of documentation as it always remains with the code. Study the API and the comments, and try out the different functions to understand what each function does and how it works.

Note that in addition to Streets, Street Segments and Intersections, the StreetsDatabaseAPI also provides 'Points of Interest' which are simply interesting landmarks, such as Union Station or a Tim Horton's store. Each 'Point of Interest' has a location, name and a type (another string) only. There are also function calls to obtain natural features like the boundaries of parks and lakes and function calls to obtain the unique OSMid (identifier) for each



a) Snapshot of Bathurst/College area from Google Maps



b) Map-database graph representation of Bathurst/College area

Figure 3: libstreetsdatabase contains a graph of intersections and street segments. You can query StreetsDatabaseAPI to find information about the graph. For example, you can find which street segments are at each intersection. Note that multiple Street Segments make up one Street – for example, the dotted Street Segments are all part of one Street (College St.).

item in the database. These OSMid values allow you to obtain additional information from the lower-level "OSMDatabaseAPI.h" functions.

```
/*
 * Copyright 2023 University of Toronto
 *
 * Permission is hereby granted, to use this software and associated
 * documentation files (the "Software") in course work at the University
 * of Toronto, or for personal use. Other uses are prohibited, in
 * particular the distribution of the Software either publicly or to third
 * parties.
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
#pragma once //protects against multiple inclusions of this header file

#include <string>
#include "LatLon.h"
#include "OSMID.h"

// Instructor-only define
#define PENALTY_FIRST
// End if instructor-pnly macros

/*******************************************************************************
 * LAYER-2 API (libstreetsdatabase) INTRODUCTION
 *
 * The libstreetsdatabase "layer 2" API provides a simplified way of interacting
 * with the OSM map data. For your convenience, we have extracted a subset of the
 * information in the OSM schema of nodes, ways, and relations with attributes and
 * pre-processed it into a form that is easier to use, consisting of familiar
 * concepts like streets, intersections, points of interest, etc. You should start
 * working with this layer first.
 *
 * The streets database is generated by the provided "osm2bin" tool, and stored in
 * a binary file called {cityname}.streets.bin.
 *
 * For access to additional feature types and attribute information, you can use
 * the underlying "layer 1" API which presents the OSM data model without
 * modification. It is more flexible but less easy to understand, however there
 * are many resources on the web including the OSM wiki and interactive online
 * viewers to help you.
 *
 * The "layer 1" API is described in OSMDatabaseAPI.h. To match objects between
 * layers, this API provides OSM IDs for all objects.
 */

```

```
// load a {map}.streets.bin file. This function must be called before any
// other function in this API can be used. Returns true if the load succeeded,
// false if it failed.
bool loadStreetsDatabaseBIN(std::string fn);

// unloads a map / frees the memory used by the API. No other api calls can
// be made until the load function is called again for some map.
// You can only have one map open at a time.
void closeStreetDatabase();



/** The extracted objects are:
 *
 * Intersections      A point (LatLon) where a street terminates, or meets one
                      or more other streets
 * Street segments    The portion of a street running between two intersections
 * Streets            A named, ordered collection of street segments running
                      between two or more intersections
 * Points of Interest (POI)   Points of significance (eg. shops, tourist
                      attractions) with a LatLon position and a name
 * Features           Marked polygonal areas which may have names (eg. parks,
                      bodies of water)
 *
 *
 * Each of the entities in a given map file is labeled with an index running from
 * 0..N-1 where N is the number of entities of that type in the map database that
 * is currently loaded. These indices are not globally unique; they depend on the
 * subset of objects in the present map, and the order in which they were loaded
 * by osm2bin.
 *
 * The number of entities of each type can be queried using getNum[...],
 * eg. getNumStreets()
 * Additional information about the i'th entity of a given type can be accessed
 * with the functions defined in the API below.
 *
 * A std::out_of_range exception is thrown if any of the provided indices are
 *  invalid.
 *
 * Each entity also has an associated OSM ID that is globally unique in the OSM
 * database, and should never change. The OSM ID of the OSM entity (Node, Way, or
 * Relation) that produced a given feature is accessible. You can use this OSMID
 * to access additional information through attribute tags, and to coordinate
 * with other OSM programs that use the IDs.
 */


/** For clarity reading the API below, the index types are all typedef'ed from
 * int. Valid street indices range from 0 .. N-1 where N=getNumStreets()
 */

typedef int FeatureIdx;
typedef int POIIdx;
typedef int StreetIdx;
```

```cpp
typedef int StreetSegmentIdx;
typedef int IntersectionIdx;

int getNumStreets();
int getNumStreetSegments();
int getNumIntersections();
int getNumPointsOfInterest();
int getNumFeatures();



/*******************************************************************************
 * Intersection
 *
 * Each intersection has at least one street segment incident on it. Each street
 * segment ends at another intersection.
 *
 * Intersection names are generated in a systematic way so that they are unique
 * in a map, but parsing them is not recommended.
 */

std::string    getIntersectionName(IntersectionIdx intersectionIdx);
LatLon         getIntersectionPosition(IntersectionIdx intersectionIdx);
OSMID          getIntersectionOSMNodeID(IntersectionIdx intersectionIdx);

// access the street segments incident on the intersection (get the count Nss
// first, then iterate through segmentNumber=0..Nss-1)
int getNumIntersectionStreetSegment(IntersectionIdx intersectionIdx);
StreetSegmentIdx getIntersectionStreetSegment(IntersectionIdx intersectionIdx, int
    segmentNumber);



/*******************************************************************************
 * Street segment
 *
 * A street segment connects two intersections. It has a speed limit, from- and
 * to-intersections, and an associated street (which has a name).
 *
 * When navigating or drawing, the street segment may have zero or more "curve
 * points" that specify its shape.
 *
 * Information about the street segment is returned in the StreetSegmentInfo
 * struct defined below.
 */

struct StreetSegmentInfo {
    OSMID wayOSMID;   // OSM ID of the source way
                      // NOTE: Multiple segments may match a single OSM way ID

    IntersectionIdx from, to;  // intersection ID this segment runs from/to
    bool oneWay;        // if true, then can only travel in from->to direction

    int numCurvePoints;      // number of curve points between the ends
    float speedLimit;        // in m/s

    StreetIdx streetID;     // index of street this segment belongs to
};

StreetSegmentInfo getStreetSegmentInfo(StreetSegmentIdx streetSegmentIdx);

// fetch the latlon of the pointNum'th curve point
// pointNum can range between 0 and StreetSegmentInfo.numCurvePoints-1
LatLon getStreetSegmentCurvePoint(StreetSegmentIdx streetSegmentIdx, int pointNum);


/*******************************************************************************
 * Street
 *
 * A street is made of multiple StreetSegments, which hold most of the
 * fine-grained information (one-way status, intersections, speed limits...).
 * The street is just a named identifier for a collection of segments.
 */

std::string getStreetName(StreetIdx streetIdx);




/*******************************************************************************
 * Points of interest
 *
 * Points of interest are derived from OSM nodes. More detailed information can be
 * accessed from the layer-1 API using the OSM ID.
 */

std::string getPOIType(POIIdx poiIdx);
std::string getPOIName(POIIdx poiIdx);
LatLon      getPOIPosition(POIIdx poiIdx);
OSMID       getPOIOSMNodeID(POIIdx poiIdx);




/*******************************************************************************
 * Natural features
 *
 * Each natural feature has a type (e.g. Park), a name (e.g. "High Park"),
 * and some number of LatLon points that define the feature. If the first
 * point (pointNum=0) and the last point (pointNum=getNumFeaturePoints-1)
 * are the same location, the feature is a closed polygon; otherwise it is a
 * polyline.
 *
 * OSM data can have degenerate, single point features.  Consider these a
 * 0 area polygon.
 * Natural features may be derived from OSM nodes, ways, or relations.
 * The TypedOSMID returned by getFeatureOSMID() can be used to match
 * features with the layer 1 API information corresponding to that OSMEntity.
 */

enum FeatureType {
    UNKNOWN = 0,
    PARK,
    BEACH,
    LAKE,
    RIVER,
    ISLAND,
    BUILDING,
    GREENSPACE,
    GOLFCOURSE,
    STREAM,
    GLACIER
};

const std::string&  getFeatureName(FeatureIdx featureIdx);
FeatureType         getFeatureType(FeatureIdx featureIdx);
TypedOSMID          getFeatureOSMID(FeatureIdx featureIdx);
int                 getNumFeaturePoints(FeatureIdx featureIdx);
LatLon              getFeaturePoint(FeatureIdx featureIdx, int pointNum);

// Calling asString on a FeatureType enumerated constant will return a string.
// This is handy for printing.
const std::string& asString(FeatureType t);
```

Listing 2: StreetsdatabaseAPI.h.

3.4 OSMDatabaseAPI.h and OSM nodes

This header file provides access to lower-level OpenStreetMap data. For this milestone, the only function you have to write that requires this lower-level data is `getOSMNodeTagValue (OSMID OSMid, std::string key)`. The first argument to this function is a very large (64-bit) integer called an OSMid that uniquely identifies an OSMNode, which is a data point on the Earth. This OSMNode can have `key,value` pairs set on it; you are to return the value (a string) associated with the requested key. For example, the OSMNode with an `OSMid of 2485404729` has a tag with key of `"name"` and a value of `"CN Tower"` – we have found the CN Tower! Listing 3 summarizes the functions you will need from OSMDatabaseAPI.h.

```
#include "OSMID.h"
#include "LatLon.h"
#include "OSMEntity.h"
#include "OSMNode.h"

/* There are three types of OSM Entities:
 *    Node      A point with lat/lon coordinates and zero
 *    Way       A collection of nodes, either a path (eg. street, bike path) or
 *              closed polygon (eg. pond, building)
 *    Relation  A collection of nodes, ways, and/or relations that share some
 *              common meaning (eg. large lakes/rivers)
 *
 * Each entity may have associated zero or more attributes of the form key=value,
 * eg. name="CN Tower" or type="tourist trap".
 */

// Load the (lower-level) layer-1 OSM database; call this before calling any other
// layer-1 API function. Returns true if successful, false otherwise.
// These files are named {cityname}.osm.bin
bool loadOSMDatabaseBIN(const std::string&);

// Close the layer-1 OSM database and release memory. You must close one map
// before loading another.
void closeOSMDatabase();


/*******************************************************************************
 * Entity access: functions to iterate over all nodes, ways, relations.
 *
 * NOTE: The indices here have no relation at all to the indices used in the
 * layer-2 API, or to the OSM IDs. You can getNodeByIndex for idx values from
 * 0 to getNumberOfNodes()-1; way and relation indices behave similarly.

 * Once you have the OSMNode/OSMWay/OSMRelation pointer, you can use it to
 * access methods of those types or the tag interface described below.
 * The most basic method supported by OSMNode, OSMWay, and OSMRelation is
 * id(), which returns the OSMID (a basic class containing a 64-bit integer).
 * The OSMID for any OSMNode/OSMWay/OSMRelation is globally unique (never
 * re-used for any other OSMNode etc. anywehere in the world).
 * For example:
 * const OSMNode *e = getNodeByIndex(0); // Gets the first OSM Node in this city
 * OSMID id = e->id();                   // A 64-bit int (big); unique id
 */

int getNumberOfNodes();      // Number of OSM nodes in this city map
int getNumberOfWays();       // Number of OSM ways in this city map
int getNumberOfRelations();  // Number of OSM relations in this city map

// Valid idx values are from 0 to getNumberOfNodes()-1 for OSM Nodes.
// OSMNode inherits from OSMEntity, so you can call OSMEntity functions with
// an OSMNode*. The most important of these is the id() function, which will
// return the OSMid of this OSMNode.
// OSMWays and OSMRelations operate similarly.
const OSMNode*          getNodeByIndex          (int idx);
const OSMWay*           getWayByIndex           (int idx);
const OSMRelation*      getRelationByIndex      (int idx);


/*******************************************************************************
 * Entity tag access
 *
 * OSMNode, OSMWay, and OSMRelation are all objects derived from OSMEntity,
 * which carries attribute tags. The functions below allow you to iterate
 * through the tags on a given entity acquired above. For example, for
 * an OSMEntity* e:
 *
 * for(int i=0;i<getTagCount(e); ++i)
 * {
 *              std::string key,value;
 *              std::tie(key,value) = getTagPair(e,i);
 *              // ... do useful stuff ...
 * }
 */

int getTagCount(const OSMEntity* e);
std::pair<std::string, std::string> getTagPair(const OSMEntity* e, int idx);

// More API functions that let you access additional OSMNode, OSMWay and
// OSMRelation data have been cut.  You can read the full header file if you
// wish to use this data later in your project, but you don't have to.
```

Listing 3: A portion of OSMDatabaseAPI.h.

3.5 Map Files

The map file for Toronto which can be loaded to use “StreetsDatabaseAPI" functions is at `/cad2/ece297s/public/maps/toronto_canada.streets.bin`; to use the lower-level “OSMDatabase API functions" you load almost the same filename, except it ends in `.osm.bin`. Maps for several other cities are also located in the `/cad2/ece297s/public/maps` directory[1].

3.6 Implement Your Own Functions

Listing 4 shows the header file for the extended API that you are going to implement in this milestone. Only the function prototypes are provided; you are to create the implementation .cpp files (you can use any set of .cpp files you wish) in which you implement each of the functions in “m1.h", plus any helper functions or classes you require. Each function prototype is commented to indicate how to use the function and what it does; use this to guide your implementation.

Comments in a header “.h" file, typically indicate how to use the function. While comments in the implementation “.cpp" file should indicate both how to use the function, and detailed comments about the implementation.

```
/*
 * Copyright 2023 University of Toronto
 *
 * Permission is hereby granted, to use this software and associated
 * documentation files (the "Software") in course work at the University
```

[1]You can generate `.bin` files for additional cities using the `osm2bin` utility which was written for this course; it converts a readable OpenStreetMap `.osm.xml` file to the `streets.bin` and `osm.bin` formats used by libstreetsdatabase.

```
 * of Toronto, or for personal use. Other uses are prohibited, in
 * particular the distribution of the Software either publicly or to third
 * parties.
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
#pragma once //protects against multiple inclusions of this header file

#include <string>
#include <vector>
#include <utility>

#include "StreetsDatabaseAPI.h"

class LatLon; //Forward declaration
class OSMID; //Forward declaration

// Use these values if you need the earth’s radius or to convert from degrees
// to radians
constexpr double kEarthRadiusInMeters = 6372797.560856;
constexpr double kDegreeToRadian = 0.017453292519943295769236907684886;


// Different functions have different speed requirements.
//
// High: this function must be very fast, and is likely to require creation of
// data structures that allow you to rapidly return the right data rather than
// always going to the StreetsDatabaseAPI or OSMDatabaseAPI
//
// Moderate: this function is speed tested, but a more straightforward
// implementation (e.g. just calling the proper streetsDatabaseAPI or
// OSMDatabaseAPI functions) should pass the speed tests.
//
// None: this function is not speed tested.


// Loads a map streets.bin and the corresponding osm.bin file and can further
// process and organize the data if you wish.
// Returns true if successful, false if an error prevents map loading.
// Speed Requirement --> moderate
bool loadMap(std::string map_streets_database_filename);

// Close the map (if loaded)
// Speed Requirement --> moderate
void closeMap();
```

```cpp

// Returns the distance between two (lattitude,longitude) coordinates in meters
// Speed Requirement --> moderate
double findDistanceBetweenTwoPoints(LatLon point_1, LatLon point_2);

// Returns the length of the given street segment in meters
// Speed Requirement --> moderate
double findStreetSegmentLength(StreetSegmentIdx street_segment_id);

// Returns the travel time to drive from one end of a street segment
// to the other, in seconds, when driving at the speed limit
// Note: (time = distance/speed_limit)
// Speed Requirement --> high
double findStreetSegmentTravelTime(StreetSegmentIdx street_segment_id);

// Returns all intersections reachable by traveling down one street segment
// from the given intersection (hint: you can't travel the wrong way on a
// 1-way street)
// the returned vector should NOT contain duplicate intersections
// Corner case: cul-de-sacs can connect an intersection to itself
// (from and to intersection on  street segment are the same). In that case
// include the intersection in the returned vector (no special handling needed).
// Speed Requirement --> high
std::vector<IntersectionIdx> findAdjacentIntersections(IntersectionIdx intersection_id);

// Returns the geographically nearest intersection (i.e. as the crow flies) to
// the given position
// Speed Requirement --> none
IntersectionIdx findClosestIntersection(LatLon my_position);

// Returns the street segments that connect to the given intersection
// Speed Requirement --> high
std::vector<StreetSegmentIdx> findStreetSegmentsOfIntersection(IntersectionIdx
    intersection_id);

// Returns all intersections along the a given street.
// There should be no duplicate intersections in the returned vector.
// Speed Requirement --> high
std::vector<IntersectionIdx> findIntersectionsOfStreet(StreetIdx street_id);

// Return all intersection ids at which the two given streets intersect
// This function will typically return one intersection id for streets
// that intersect and a length 0 vector for streets that do not. For unusual
// curved streets it is possible to have more than one intersection at which
// two streets cross.
// There should be no duplicate intersections in the returned vector.
// Speed Requirement --> high
std::vector<IntersectionIdx> findIntersectionsOfTwoStreets(StreetIdx street_id1,
    StreetIdx street_id2);

// Returns all street ids corresponding to street names that start with the
// given prefix
// The function should be case-insensitive to the street prefix.
// The function should ignore spaces.
//  For example, both "bloor " and "BloOrst" are prefixes to
// "Bloor Street East".
// If no street names match the given prefix, this routine returns an empty
// (length 0) vector.
// You can choose what to return if the street prefix passed in is an empty
// (length 0) string, but your program must not crash if street_prefix is a
// length 0 string.
// Speed Requirement --> high
std::vector<StreetIdx> findStreetIdsFromPartialStreetName(std::string street_prefix);

// Returns the length of a given street in meters
// Speed Requirement --> high
double findStreetLength(StreetIdx street_id);

// Returns the nearest point of interest of the given type (e.g. "restaurant")
// to the given position
// Speed Requirement --> none
POIIdx findClosestPOI(LatLon my_position, std::string POItype);

// Returns the area of the given closed feature in square meters
// Assume a non self-intersecting polygon (i.e. no holes)
// Return 0 if this feature is not a closed polygon.
// Speed Requirement --> moderate
double findFeatureArea(FeatureIdx feature_id);

// Return the value associated with this key on the specified OSMNode.
// If this OSMNode does not exist in the current map, or the specified key is
// not set on the specified OSMNode, return an empty string.
// Speed Requirement --> high
std::string getOSMNodeTagValue (OSMID OSMid, std::string key);
```

Listing 4: Milestone 1 API “m1.h".

3.6.1 Computing Distance from Latitude/Longitude

In this and the following subsections, some of the functions in StreetsDatabaseAPI.h are clarified.

Locations in libstreetsdatabase are represented as latitude and longitude, in degrees. There are many ways to compute the distance between two latitude/longitude (lat/lon) points; some more accurate than others. We will be using Pythagoras’ theorem on an equirectangular projection[2].

To compute (x,y) coordinates on the surface of the Earth (in m) from lat/lon, use the equation below:

$$(x, y) = (R \cdot lon \cdot cos(lat_{avg}),\ R \cdot lat) \tag{1}$$

[2]A map projection is a translation from lat/lon to (x,y) coordinates. The projection basically draws out planet (which is almost spherical) on a rectangular coordinate system. If you are interested you can read more about map-making and more accurate projections at http://en.wikipedia.org/wiki/List_of_map_projections.

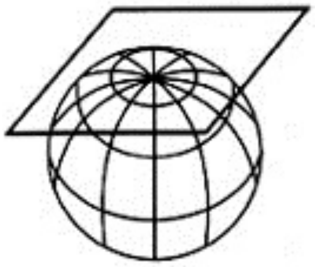


Figure 4: Projecting part of the surface of the earth to a flat plane. Source: hosting.soonet.ca/eliris/gpsgis/Lec2Geodesy.html.

where *lat* and *lon* are in *radians*, lat_{avg} is the average latitude of the area being mapped and R is the radius of the Earth. The equation above accounts for the fact that the distance between lines of latitude is the same everywhere on earth, while the distance between lines of longitude depends on how far away from the equator you are. At the equator (latitude = 0 degrees) the distance between two lines of longitude is equal to that between two lines of latitude. At the North or South Pole (latitude = 90 or -90 degrees, respectively) all the lines of longitude converge to a point so there is no distance between them. We have defined a constant for the radius of the Earth in `m1.h` so you can implement the formula above precisely and match the autotester.

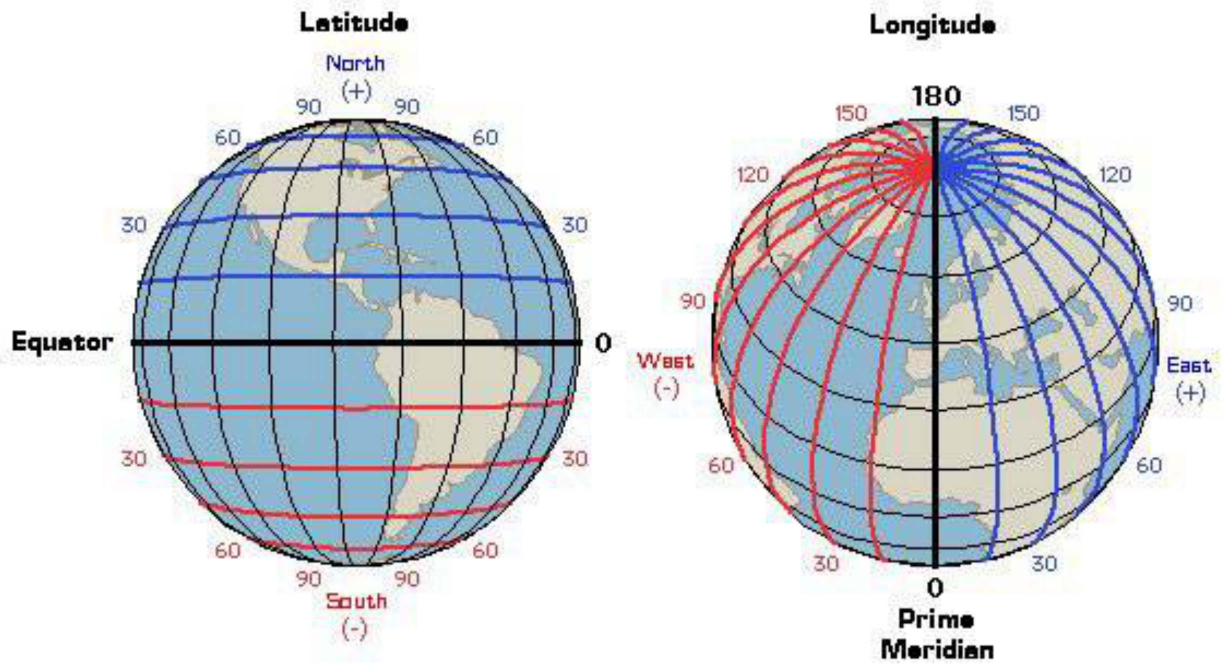


Figure 5: Parallels of latitude and lines of longitude. Source: blog.eogn.com/2014/09/16/convert-an-address-to-latitude-and-longitude/.

If we use the (x, y) projection above to draw our map (as we will in milestone 2), then lat_{avg} is the mid-latitude of the map we're drawing – i.e. the average between the min/max latitudes of the city boundaries $\frac{lat_{min}+lat_{max}}{2}$. However, if we are using the projection to find the distance between two points (lon_1, lat_1) and (lon_2, lat_2) then it is more accurate to compute lat_{avg} as $\frac{lat_1+lat_2}{2}$. This is the lat_{avg} that you should be using to compute distance, and the autotester uses the same equation to verify your answers.

To find the distance between two points, we convert (lon, lat) to (x, y) locations in m as described above, and then use Pythagoras' theorem:

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2} \tag{2}$$

3.6.2 Curve Points

Not all street segments are perfectly straight lines between two intersections; some city blocks (street segments) are curved or even winding. Curve points are properties of street segments that allow the StreetsDatabaseAPI to represent such curved or winding roads between intersections. As Fig. 6 shows, some street segments have no curve points (they are perfectly straight); the distance one must travel along the street segment between the two intersections can therefore be computed from the (Latitude/Longitude) locations of the two intersections. Other street segments have curve points, and each one is represented as a Latitude/Longitude pair. Therefore, to find the distance along such a street segment these curve points must be taken into account to find the correct driving distance.

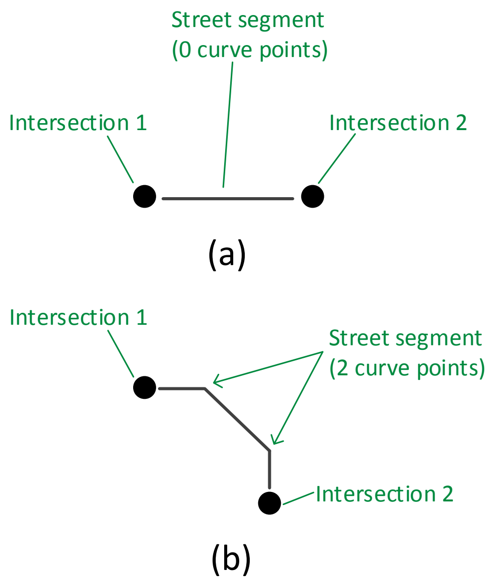


Figure 6: A street segment exists between two intersections; in (a) the street segment has no curve points, while in (b) the street segment has 2 curve points. Note that you can get the latitude/longitude of each curve point or intersection.

3.6.3 Using the Autotester and Unit Tests

An important part of any software project is verifying that the code is correct. One popular and effective approach is called *Unit Testing*. With unit testing you verify the correctness of small 'units' of your code, such as individual functions or classes. Testing at such a low level is beneficial since errors are much easier to detect and debug since they are isolated to a small subset of your program.

~30mins

To learn more about Unit Testing see the "ECE297 Quick Start Guide: Unit Testing"

We have written several unit tests that check your milestone 1 API functions for both correctness and run-time performance. You can run these tests with the `ece297exercise` command, and it will summarize how many of the test cases pass or fail and how fast your functions run compared to the speed target.

To get full marks you need to implement all functions, pass all testcases and meet the speed targets. Note that autotester (`ece297exercise`) only exercises the public testcases which are not exhaustive. For grading, we will be testing your code with more (private) testcases as well so make sure your implementation handles any corner cases. Note that while you can use your main function to test your code, the main function will not be graded for this milestone.

Use the autotester to test your implementation with public testcases for correctness and performance. Additional private testcases will also be used for grading.

The autotester can be run using the '`ece297exercise 1`' command:

```
> ls #In the main project directory
Makefile  build  libstreetmap  libstreetmap.a  main  mapper  nbproject  test_libstreetmap


> ece297exercise 1 #Runs the autotester
The following 10 tester(s) will be run:
        M1_Func_Intersection_Tests
        M1_Func_Street_Tests
        M1_Func_Distance-Time_Tests
        M1_Func_Spatial_Tests
        M1_Perf_Intersection_Tests
        M1_Perf_Street_Tests
        M1_Perf_Distance-Time_Tests
        M1_Perf_Spatial_Tests
        M1_Load_Maps
        Valgrind

Running Tester: M1_Func_Intersection_Tests
  Building   M1_Func_Intersection_Tests
#Output trimmed...
```

```
Test Summary: PASS ( 0 of 31 failed)
  UnitTests PASS ( 0 of 31 failed)
```

Listing 5: Exercise example

The `ece297exercise` command runs the program executable in your current directory. For speed tests, you should make sure you are running the Release Configuration of your program (i.e. that you built the Release Configuration before running `ece297exercise`), as that is the fastest version of your program and that is how we will test your submission. As well, note that if your machine is heavily loaded by either you or other students running multiple CPU-intensive programs at the same time it can slow down your program. We will test your submission on an unloaded machine.

If you wish, you can tell ece297exercise to run only a certain tests; this can be useful if you're trying to debug one function and don't want to wait for all the tests to run. Use `ece297exercise -h` for help on the options you can specify.

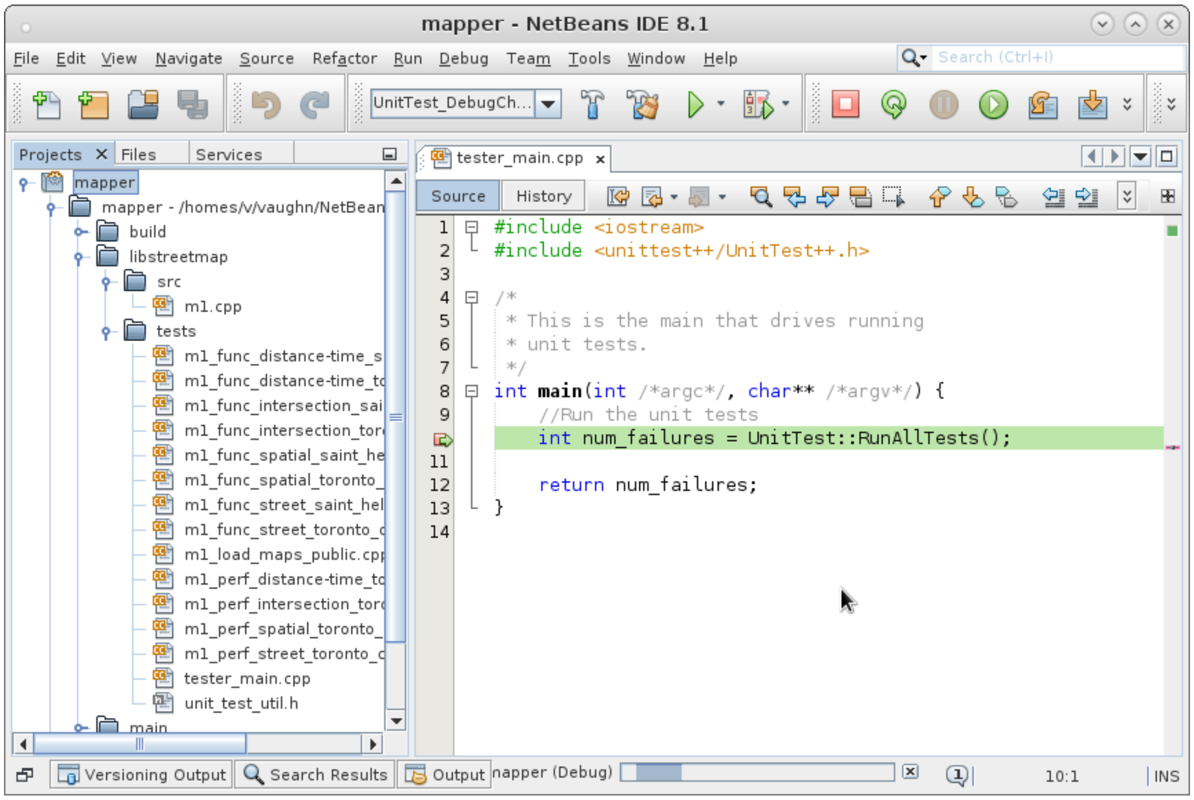
The unit tests we have provided in ece297exercise for this milestone will only test your API with valid input, including corner case inputs where the m1.h header file we provide clearly indicates what should be returned if no valid result exists. Integer indices will always be within the valid range for that type of data, e.g. the argument will be between 0 and `getNumIntersections() -1` for intersection indices. You may wish to make your API test for invalid input (e.g. intersection indices that are out of range) and take some appropriate action (like printing an error message) since this will make your API more robust, and you will be building code that uses your API in the later milestones in this program. Note that while the public tests in ece297exercise test many aspects of your API, they usually will not test all the corner cases and possible valid inputs to your functions. You should ensure that your code covers all cases, as the private tests may test additional cases.

3.7 Debugging Unit Tests

You can debug any unit tests you have in your project (files in libstreetmap/tests) by choosing a UnitTest configuration, such as `UnitTest_DebugCheck`, in NetBeans or another IDE, then building the code and starting the debugger as shown in Figure 7.

If you want to debug a unit test run by the `ece297exercise` command, execute the commands below to copy the unit tests to your project (working directory).

```
> ls  #In the project directory
Makefile  build   libstreetmap   libstreetmap.a   main   nbproject
%> cp /cad2/ece297s/public/m1/tests/* libstreetmap/tests
> ece297exercise 1 --list_testers
Supported Milestone 1 Testers:
   Public_M1_Func_Intersection_Tests
   Public_M1_Func_Street_Tests
         ... More testers ...
> ece297exercise 1 --debug_tester Public_M1_Func_Street_Tests  # Pick any test
Backing up the already existing tests to libstreetmap/tests/.old_tests
Copying the required test files
DONE!
```



```
#include <iostream>
#include <unittest++/UnitTest++.h>

/*
 * This is the main that drives running
 * unit tests.
 */
int main(int /*argc*/, char** /*argv*/) {
    //Run the unit tests
    int num_failures = UnitTest::RunAllTests();

    return num_failures;
}
```

Figure 7: Debugging unit tests in NetBeans.

```
==========================
Recompile the project in your IDE using UnitTest_* configurations
  To debug, open the debugger in your IDE
```

Listing 6: Copying the a public unit test run by ece297exercise into your project

For speed testing, you will want to build the UnitTest_Release configuration and then run without the debugger. If you prefer to work at the command line, you can also type `make test` from a terminal in the project directory.

If your project passes all unit tests the output will indicate success:

```
> make test
#Output trimmed...
Success: 15 tests passed.
Test time: 0.07 seconds.
```

Listing 7: Unit testing usage

If your project fails unit tests the output will indicate the error and where it occurred:

```
> make test #We have a bug in our program so this will error
#Output trimmed...
libstreetmap/tests/m1_unittests.cpp:514: error: Failure in
    directly_connected_functionality: !are_directly_connected(1358, 2709)
```

```
FAILURE: 1 out of 15 tests failed (1 failures).
Test time: 0.09 seconds.
make: *** [test] Error 1
```

Listing 8: Unit testing usage

3.7.1 Adding a Unit Test

You can (but are not required to) add additional unit tests to further test functions in `m1.h` or functions you have created for internal use which are not exposed in `m1.h`. If you add other unit tests for these functions show then to your TA, and they will be considered for extra credit when assigning grades.

3.8 Grading

3.8.1 Submitting Your Code

Submit your project using the '`ece297submit 1`' command:

```
 > ls #In the main project directory (must be in git)
Makefile  build  libstreetmap  libstreetmap.a  main  mapper  nbproject  test_libstreetmap

> ece297submit 1 #Submits the project
#Output trimmed...
Committing Submission
Verified submitted file exists and matches size
Successfully committed submission.
```

Listing 9: Submission example

ece297submit submits the latest (head) revision of your project from your git remote repository. Make sure you **commit** and **push** before running ece297submit so your latest code is submitted.

3.8.2 Grading Scheme

You are required to implement all the functions that are specified in "m1.h". In doing so you should also make sure that you comment your code well and commit to your git repository often; you should also use informative git commit messages as your TA will look at the git log. There are 10marks assigned to this milestone which divided as follows:

- 7 marks: Code functionality (4 marks) and runtime (3 marks). Assessed by the autotester on both public and private test cases.
- 3 marks: Project management, organization and wiki; effective use of git, code style and commenting. Assessed by your TA. You should also be able to answer questions about how your code works and why it is structured the way it is. A rubric giving more information is available on quercus.

Note that different team members may receive different marks based on the clarity of their answers to questions, their contribution to the milestone as shown by the wiki and git logs, and their knowledge of the code.