Osmium Library Manual

October 29, 2014

Contents

1 Introduction

Simple things should be simple and complex things should be possible. - Alan Kay

The OpenStreetMap project is growing at an enormous rate. Working with the OSM data becomes increasingly difficult, because there is just so much of it and because it gets more complex all the time.

Osmium was developed as an answer to this challenge. After year of developing software to work with OSM data in many programming languages like Perl, Ruby, Java and even in XSLT, it became evident that something more was needed to efficiently work with these huge amounts of data. Processing speed was, of course, one big issue here, but the other one is available memory. Data processing tasks can be so much faster if their working set fits into memory, that it makes sense to think about this. Because Osmium is a C++ library it can make very efficient use of the main memory on your computer. Primitive objects such as integers and doubles, but also complex objects need only as much memory as is really necessary. There is not much management overhead needed in many cases, if the data structures are chosen carefully.

Osmium has been in continuous development since it was borne in October 2010. And it has changed considerably over time. While the basic premise, to write a low-level efficient OSM library, is still true, it has become more and more powerful and at the same time easier to use. Osmium has been in production use nearly from day one, some parts of it have been ripped from earlier production code. Osmium is not an academic exercise, but it is used and it has shown its power many times. And while C++ might not be the easiest programming language to learn and Osmium might not be the easiest library to use, we try to make it as simple as possible to work with it, as long as this doesn't compromise efficiency too much.

1.1 Header-only Library

Osmium is a header-only library, so there is nothing to compile to build it. Just include the header files you need.

1.2 The `osmium` Namespace

Everything in the Osmium library is in the `osmium` namespace or in sub-namespaces. You'll likely encounter the `osmium::io` namespace for everything related to Input and Output and the `osmium::geom` namespace for geometry-related functionality, but there are some more. Do not directly use anything in any sub-namespace called `detail`. Those classes and functions are for internal use only.

1.3 C++11

The current version of Osmium makes extensive use of the new facilities provided by the C++11 standard. It will only work with current compilers and libraries.

1.4 License

The Osmium Library is available under the very liberal Boost Software License:

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

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, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

This manual is available under the Creative Commons Attribution-ShareAlike License version 4.0.

1.5 Old Versions of Osmium

If you are stuck with older compilers or need to develop using the older C++98 for some reason, you can have a look at the older Osmium version. But it is quite different from the current Osmium described in this manual.

The old Osmium is available from GitHub.

2 Compiling Programs Using Osmium

Osmium is a header-only library, so it does not need to be compiled by itself. But of course you will have to compile programs using Osmium. This chapter gives you some hints on how to best do this.

2.1 C++11

Osmium uses modern standard C++11 and should compile everywhere without warnings when you have a recent enough compiler and standard library. It works with GCC 4.8 and clang 3.2 or newer. Some parts might work with GCC 4.7 or older versions of clang.

You might have to set the C++ version using the compiler option

```
-std=c++11
```

2.2 Operating Systems

Linux Osmium is developed on Linux and tested best on that system. Debian Jessie (testing) and current Ubuntu systems come with everything needed for Osmium. Debian wheezy (stable) and the Ubuntu LTS release 12.04 don't have compilers current enough. If you are stuck on these systems, use a backported compiler.

Mac OSX Osmium also works well on Mac OSX with the exception of the parts that need the mremap system call that is not available on Mac OSX.

Windows Most of Osmium should work on Windows, but nobody has been actually testing it.

2.3 Build System

The example programs in the example directory use a simple Makefile to help compiling them. Osmium is simple enough that it doesn't need a large build systems like automake or cmake to compile programs using it. This means that it works with whatever build system you are using in your programs.

2.4 Large File Support

When working with OSM data you often have very large files with several gigabytes. This can lead to problems on 32bit systems. Use the options

```
-D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64
```

for the compiler to make sure that large files work.

2.5 Include What You Need

Libosmium is a header-only library. You include those parts you need and ignore the rest. The library makes sure internal dependencies are followed, of course. Whatever you don't include doesn't bloat your program and it doesn't need execution time. And, what's more important, you do not have dependencies on external libraries that are not really needed for whatever you are doing.

The full Osmium library has a frightening array of dependencies, but thats because its mission is to make OSM data usable with all those libraries. But if you don't need some part of Osmium, you also will not need those libraries.

You will find information about which files to include in the different chapters of this manual. Generally for a class `osmium::foo::Bar`, there will be an include file called `<osmium/foo/bar.hpp>` that you need to include, but sometimes it is a bit more difficult.

Do not directly include any header files in directories called "detail". They are for internal use of the library only.

2.6 Boost

Some parts of Osmium need some Boost libraries. Libraries used are:

- Operators
- Iterator
- String Algorithms

Osmium also needs some Boost unicode functions. Because they are rather new and not available everywhere, they are currently included with Osmium in `include/boost_unicode_iterator.hpp`.

The Osmium unit tests use the Boost Test library.

2.7 OGR Support

Osmium can create OGR geometries from OSM data. To use this compile with what the command

```
gdal-config --cflags
```

returns and link with what

```
gdal-config --libs
```

returns.

3 Basic Types

All the types and classes described in this chapter are value types, ie they are small and can be copied around cheaply.

3.1 IDs

Typedef: `osmium::object_id_type`

Include: `<osmium/osm/types.hpp>`

For object IDs use the type `osmium::object_id_type`. It is a 64bit signed integer that can represent the more than 2 billion nodes we already have in OSM. While way and relation IDs could theoretically use a smaller ID type (signed 32 bit are currently enough), for consistency and to be future-proof, they will also use this type in most cases.

OSM objects always have positive IDs. But some software (such as JOSM) uses negative IDs for objects that have not yet been uploaded to the main OSM database. To support these use cases, the `object_id_type` is a signed integer.

Some parts of Osmium, notably the different index classes, can only work with positive IDs. In those cases the type `osmium::unsigned_object_id_type` is used. If you know that your data only contains positive IDs or only negative IDs, you can use the `positive_id()` member function on the `Object` class to get IDs of that type. It will return the absolute value of the ID.

If your data contains a mix of positive and negative IDs, this simple approach will fail! In that case you have to use two indexes, one for the positive IDs and one for the negative IDs. The `osmium::handler::NodeLocationsForWay` class takes this approach.

3.2 Other Primitive Types

Include: `<osmium/osm/types.hpp>`

There are several other typedefs:

Type	Description
`object_version_type`	type for OSM object version number
`changeset_id_type`	type for OSM changeset IDs
`user_id_type`	type for OSM user IDs
`num_changes_type`	type for changeset num_changes

All these types are currently 32bit integers. Version numbers, changeset IDs and User IDs are always positive (they start out with 1). The number of changes can be 0 or larger.

3.3 Locations

Class: `osmium::Location`

Include: `<osmium/osm/location.hpp>`

In Osmium all positions on Earth are stored in objects of the osmium::Location class. Coordinates are stored as 32 bit signed integers after multiplying the coordinates with `osmium::coordinate_precision` = 10,000,000. This means we can store coordinates with a resolution of better than one centimeter, good enough for OSM use. The main OSM database uses the same system. We do this to save memory, a 32 bit integer uses only 4 bytes, a double uses 8.

Coordinates are not checked when they are set.

To create a location:

```
osmium::Location location(9.3, 49.7);
```

or using integers:

```
osmium::Location location(9300000000, 49700000000);
```

Make sure you are using the right number type or you will get very wrong coordinates.

You can also create an undefined location. This is used for instance for coordinates in ways that are not set yet:

```
osmium::Location location();
```

In a boolean context an undefined location returns false, a defined true. So you can write something like:

```
if (location) {
    ...defined location here...
}
```

You can get and set the coordinates using the internal (integer) format with the `x()` and `y()` member functions and the external (double) format with the `lon()` and `lat()` member functions.

The normal bounds for the longitude and latitude are -180 to 180 and -90 to 90, respectively. But in historic OSM data you can sometimes find locations outside these bounds. Call

```
location.valid()
```

to find out if a location is inside those bounds.

The `lon()` and `lat()` getter calls will throw an exception if the location is invalid or undefined.

3.4 Segments

Class: `osmium::Segment`

Include: `<osmium/osm/segment.hpp>`

Segments are the directed connection between two locations. They are not OSM objects but sometimes useful in algorithms.

3.5 Undirected Segments

Class: `osmium::UndirectedSegment`

Include: `<osmium/osm/undirected_segment.hpp>`

Undirected Segments are connection between two locations. They are not OSM objects but sometimes useful in algorithms.

3.6 Boxes

Class: `osmium::Box`

Include: `<osmium/osm/box.hpp>`

A box is a rectangle described by the minimum and maximum longitude and latitude. It is used, for instance, in the header of OSM files and in changesets to describe the bounding box.

```
osmium::Box box;
box.extend(osmium::Location(3.2, 4.3));
box.extend({4.5, 7.2});
box.extend({3.3, 8.9});
std::cout << box;  // (3.2,4.3,4.5,8.9)
```

4 OSM Entities

Osmium works with the four basic types of OSM entities: Nodes, Ways, and Relations (which are all OSM Objects) and Changesets. In addition Areas are supported, which are not native OSM objects, but they are almost treated like real OSM objects.

These OSM entities can not be created like any normal C++ object, but they need a buffer to live in. See the next chapter for details. Accessing existing OSM entities on the other hand is easy and straightforward.

4.1 OSM Objects

Class: `osmium::OSMObject`

Include: `<osmium/osm/object.hpp>`

The `osmium::OSMObject` class is the base class for nodes, ways, and relations. it has accessors for the usual OSM attributes:

```
osmium::OSMObject& obj = ...
std::cout << "id=" << obj.id()
          << " version=" << obj.version()
          << " timestamp=" << obj.timestamp()
          << " visible=" << (obj.visible() ? "true" : "false"
          << " changeset=" << obj.changeset()
          << " uid=" << obj.uid()
          << " user=" << obj.user() << "\n";
```

The `changeset()` and `uid()` accessor functions return the IDs of the changeset that created this object version and the User ID of the user creating this version of the object, respectively. They do not link to an object of that type.

The `visible` flag will always be true for normal OSM data, but for history data or change files it shows whether an object version has been deleted.

In addition each object has a list of tags attached:

```
const osmium::TagList& tags = obj.tags();
```

You can iterate over all tags:

```
for (const osmium::Tag& tag : obj.tags()) {
    std::cout << tag.key() << "=" << tag.value() << "\n";
}
```

Or you can find specific tags:

```
const char* highway = obj.tags().get_value_by_key("highway");
if (highway && !strcmp(highway, "primary") {
    ...
}
```

4.2 Nodes

Class: `osmium::Node`

Include: `<osmium/osm/node.hpp>`

A `Node` is a kind of `OSMObject`. In addition to the things you can do with any OSMObject, the Node has a Location.

```
osmium::Node& node = ...
double longitude = node.location().lon();
```

4.3 Ways

Classes: `osmium::Way`, `osmium::WayNode`, `osmium::WayNodeList`

Include: `<osmium/osm/way.hpp>`

A `Way` is a kind of `OSMObject`. In addition to the things you can do with any OSMObject, a Way has a list of node references:

```
osmium::Way& way = ...
for (const osmium::NodeRef& nr : way.nodes()) {
    std::cout << "ref=" << nr.ref() << " location=" << nr.location() << "\n";
}
```

4.4 Relations

Classes: `osmium::Relation`, `osmium::RelationMember`, `osmium::RelationMemberList`

Include: `<osmium/osm/relation.hpp>`

A `Relation` is a kind of `OSMObject`. In addition to the things you can do with any OSMObject, a Relation has a list of members:

```
osmium::Relation& relation = ...
osmium::RelationMemberList& rml = way.members();
for (osmium::RelationMember& rm : rml) {
    std::cout << rm.type() << rm.ref() << " (role=" << rm.role() << ")\n";
}
```

4.5 Areas

not yet documented

4.6 Changesets

Class: `osmium:Changeset`

Include: `<osmium/osm/changeset.hpp>`

Changesets contain the metadata for a set of changes to OSM data.

```
osmium::Changeset
```

5 Buffers

OSM entities have to be stored somewhere in memory. They are complex objects containing arbitrary number of tags, relations can have any number of members etc. If we handled those objects like any normal C++ object, creating them would take lots of small memory allocations and many pointer indirections to get at all the parts of the data. Instead OSM entities are created inside so-called *buffers*. Buffers can have a fixed size or grow as needed. New objects can be added at the end, and they are stored inside those buffers in a reasonably space-efficient manner while still being accessible easily and quickly.

Buffers can be moved around between different parts of your program and even between threads. The content of buffers can even be written to disk as it is and read back in and immediately used "as is" without any serializaton or de-serialization step needed.

But all of this has one draw-back: It is slightly more complicated to create those objects and they can not just be instantiated on the stack.

Buffers can not be copied, because it is unclear who would be responsible for the memory then. But they can be moved.

5.1 Creating a Buffer

Buffers exist in two different flavours, those with external memory management and those with internal memory management. If you already have some memory with data in it (for instance read from disk), you create a Buffer with external memory managment. It is your job then to free the memory once the buffer isn't used any more. If you don't have some memory space already, you can create a Buffer object and have it manage the memory internally. It will dynamically allocate memory and free it again after use.

To create a buffer from existing memory you give the address and size to the constructor:

```
const int buffer_size = 10240;
void* mem = malloc(buffer_size);
osmium::memory::Buffer buffer(mem, buffer_size);
```

This will create an empty buffer with `buffer_size` bytes available for use.

If the new buffer already contains some data, you can add the number of bytes already in use as a third parameter to the constructor:

```
void* mem = malloc(buffer_size);
int num = read(0, mem, buffer_size);
osmium::memory::Buffer buffer(mem, buffer_size, num);
```

To create a buffer with internal memory-management you construct it with the number of bytes it should have initially and a flag that tells Osmium whether it should automatically grow the buffer if it is needed:

```
const int buffer_size = 10240;
osmium::memory::Buffer buffer(buffer_size, osmium::memory::Buffer::auto_grow::yes);
osmium::memory::Buffer buffer(buffer_size, osmium::memory::Buffer::auto_grow::no);
```

5.2 Adding Items to the Buffer

5.3 Handling a Full Buffer

If a buffer becomes full, there are three different things that can happen:

If the buffer was created with `auto_grow::yes`, it will reserve more memory on the heap and double its size. This will happen without the client code noticing, but it will invalidate any pointer pointing into the buffer. This is the same behaviour a `std::vector` has so it should be familiar to C++ programmers.

If the buffer was created with `auto_grow::no` (or if it is a buffer with external memory management), the exception `osmium::memory::BufferIsFull` will be thrown. In this case you have to catch the exception, either grow the buffer or create a new one. If you grow the buffer you can keep going at the point where you left off. If you start a new one, the last object you were writing to the buffer when the exception was thrown was not committed and you have to write it again into the new buffer.

As a third option you can set a *callback* functor that wil be called when the buffer is full. The functor takes a reference to the buffer as argument and returns void:

```
void full(osmium::memory::Buffer& buffer) {
    std::cout << "Buffer is full\n";
}

osmium::memory::Buffer buffer(buffer_size, false);
buffer.set_full_callback(full);
```

6 Input and Output

Most programs using OSM data will need to read from OSM files and/or write to OSM files. Osmium supports several different OSM file formats and has many different ways of accessing the data in convenient ways.

6.1 File Formats

Osmium supports the following formats:

XML The original XML-based OSM format. This format is rather verbose and working with it is slow, but it is still used often and in some cases there is no alternative. The main OSM database API also returns its data in this format. More information about this format on the OSM Wiki.

PBF The binary format based on the Protobuf library. This is the most compact format. More information on the OSM Wiki.

OPL A simple format similar to CSV-files with one OSM entity per line. This format is intended for easy use with standard UNIX command line tools such as `grep`, `cut`, and `awk`.

See Output Formats for more details about these formats.

6.2 Compression

Osmium supports compression and decompression of XML and OPL files internally using the GZIP and BZIP2 formats. If you want to use compression you have to include the right header files and link to the `libz` and `libbz2` libraries, respectively.

6.3 Headers

Whenever you want to use Osmium to access OSM files you need to include the right header files and link your program to the right libraries. If you want to support all the different formats you add

```
#include <osmium/io/any_input.hpp>
```

and/or

```
#include <osmium/io/any_output.hpp>
```

to your C++ files. These headers will pull in all the file formats and all the compression types for input and output, respectively. Usually this is what you want to use. But if you are sure you don't need all formats or if you don't have all the libraries needed for all the formats, you can pick and choose formats and compression types.

If you only need some file formats, you can include any combinations of the following headers:

```
#include <osmium/io/pbf_input.hpp>
#include <osmium/io/xml_input.hpp>

#include <osmium/io/pbf_output.hpp>
#include <osmium/io/opl_output.hpp>
#include <osmium/io/xml_output.hpp>
```

If you want compression support, you have to add the includes for the different compression algorithms:

```
#include <osmium/io/gzip_compression.hpp>
#include <osmium/io/bzip2_compression.hpp>
```

Or, if you want both anyway, you can just use the shortcut:

```
#include <osmium/io/any_compression.hpp>
```

6.4 Output Formats

6.4.1 XML

There are several different XML formats in use in the OSM project. The main formats are the one used for planet files, extracts, and API responses (suffix `.osm`), the format used for change files (suffix `.osc`) and the history format (suffixes `.osm` or `.osh`).

Some variants are also used, such as the JOSM format which is similiar to the normal OSM format but has some additions. Support for the features of these formats varies.

When reading, the OSM change format (`.osc`) is detected automatically. When writing, you have to set it using the format specifier `osc` or the format parameter `xml_change_format=true`.

For read support you need the expat parser library. Link with:

```
-lexpat
```

For write support no special library is needed.

6.4.2 PBF

The PBF file format is based on the Google Protocol Buffers library. PBF files are very space efficient and faster to use than XML files. PBF files can contain normal OSM data or OSM history data, but there is no equivalent to the XML .osc format.

The OSM PBF format is defined in libosmpbf, you'll probably have to compile and install this yourself before using it in Osmium.

To build with PBF support, several libraries are needed: libprotobuf-lite contains the Protocol Buffers library itself which also needs libpthreads, for compression libz is needed. Those are all standard libraries that should be available on most systems.

To summarize, you need to link with:

```
-pthread -lprotobuf-lite -losmpbf -lz
```

The Google Protocol Buffers library allocates some global buffer memory which is never freed. You can call the following function in your code to free these buffers:

```
google::protobuf::ShutdownProtobufLibrary();
```

You do not have to do this, the function is not necessary for the correct functioning of your program. But if you are using a memory checker like Valgrind you will get error messages otherwise.

Osmium supports reading and writing of nodes in *DenseNodes* and non-*DenseNodes* formats. Default is *DenseNodes*, as this is much more space-efficient. Add the format parameter `pbf_dense_nodes=false` to disable *DenseNodes*.

Osmium usually will compress PBF blocks using zlib. To disable this, use the format parameter `pbf_compression=none`.

Usually PBF files contain all the metadata for objects such as changeset id, username, etc. To save some space you can disable writing of metatdata with the format parameter `pbf_add_metadata=false`.

6.4.3 OPL ("Object Per Line") Format

This format is preliminary, it might change. Please send feedback if you use this format!

This format was created to allow easy access to and manipulation of OSM data with typical UNIX command line tools such as `grep`, `sed`, and `awk`. This can make some ad-hoc OSM data manipulation easy to do, but is probably not as fast as some specialized tool. But it beats grepping in XML files...

OPL files are only about half the size of OSM XML files, but when compressed they are about the same size.

Osmium currently can write OPL files, but not read them.

Each line of the file contains one OSM object (a node, way, or relation) or an OSM changeset. Fields are separated by a space character, lines by a newline character. Fields always appear in the same order and are always present, each field is introduced by a specific character:

One of these fields is always the first:

```
n - Node ID (nodes only)
w - Way ID (ways only)
r - Relation ID (relations only)
c - Changeset ID (changesets only)
```

Then for OSM objects in the given order:

```
v - Version
d - Deleted flag ('V' - visible or 'D' - deleted)
c - Changeset ID
t - Timestamp (ISO Format)
i - User ID
u - Username
T - Tags
x - Longitude (nodes only)
y - Latitude (nodes only)
N - Nodes (ways only)
M - Members (relations only)
```

The N, M, and T fields can be empty. If the user is anonymous, the ‘User ID’ will be 0 and the ‘Username’ field will be empty: `...  i0 u ....` If the node is deleted, the ‘Longitude’ and ‘Latitude’ fields are empty. All other fields always contain data.

The ‘Deleted flag’ shows whether an object version has been deleted (`dD`) or whether it is visible (`dV`). For normal OSM data files this is always `dV`, but change files and osm history files can contain deleted objects.

For changesets the fields are different:

```
k - num_changes
s - created_at (start) timestamp (ISO Format)
e - closed_at (end) timestamp (ISO Format)
i - User ID
u - Username
x - Longitude (left bottom corner, min_lon)
y - Latitude (left bottom corner, min_lat)
X - Longitude (right top corner, max_lon)
Y - Latitude (right top corner, max_lat)
T - Tags
```

The fields e is empty when the changeset is not closed yet. The fields x, y, X, Y can be empty when no bounding box could be derived. The field k can be 0.

Escaping

User names, tags, and relation member roles can contain any valid Unicode character. Any characters that have special meaning in OPL files (‘’ (space), ‘,’ (comma), ‘=’ (equals) and ‘@’) have to be escaped as well as any non-printing characters.

Escaped characters are written as `%xxxx`, ie a percent sign followed by the 4-digit hex code of the Unicode code point.

Currently there is a hard-coded list in the code of all the characters that don’t need escaping. This list is incomplete and subject to change.

Format Overview

(Some lines have been broken in this description for easier reading, in the file format they are not.)

```
NODE:
    n(OBJECT_ID) v(VERSION) d(V|D) c(CHANGESET_ID) t(yyyy-mm-ddThh:mm:ssZ)
    i(USER_ID) u(USERNAME) T(TAGS) x(LONGITUDE) y(LATITUDE)

WAY:
    w(OBJECT_ID) v(VERSION) d(V|D) c(CHANGESET_ID) t(yyyy-mm-ddThh:mm:ssZ)
    i(USER_ID) u(USERNAME) T(TAGS) N(WAY_NODES)

RELATION:
    r(OBJECT_ID) v(VERSION) d(V|D) c(CHANGESET_ID) t(yyyy-mm-ddThh:mm:ssZ)
    i(USER_ID) u(USERNAME) T(TAGS) M(MEMBERS)

CHANGESET:
    c(CHANGESET_ID) k(NUM_CHANGES) s(yyyy-mm-ddThh:mm:ssZ) e(yyyy-mm-ddThh:mm:ssZ)
    i(USER_ID) u(USERNAME) x(LONGITUDE) y(LATITUDE) X(LONGITUDE) Y(LATITUDE) T(TAGS)

TAGS
    (KEY)=(VALUE),...

WAY_NODES:
    n(NODE_REF),...

MEMBERS:
    [nwr](MEMBER_REF)@(MEMBER_ROLE),...
```

Usage Examples

(Note that some of these commands generate quite a lot of output. You might want to add a `| less` or redirect into a file. For larger OSM files some of these commands might take quite a while, so try them out on small files first.)

Find all objects tagged `highway=...`:

```
egrep "( T|,)highway=" data.osm.opl
```

Find all IDs of ways tagged `highway=...`:

```
egrep '^w' data.osm.opl | egrep "( T|,)highway=" | cut -d' ' -f1 | cut -c2-
```

Find all nodes with version > 9:

```
egrep '^n' data.osm.opl | egrep -v ' v. '
```

Find the first fields of the relation with the highest version number:

```
egrep '^r' data.osm.opl | sort -b -n -k 2.2,2 | tail -1 | cut -d' ' -f1-7
```

Find all objects with changeset ID 123:

```
egrep ' c123 ' data.osm.opl
```

Count how many objects were created in each hour of the day:

```
egrep ' v1 ' data.osm.opl | cut -d' ' -f5 | cut -dT -f2 | cut -d: -f1 | sort | uniq -c
```

Find all closed ways:

```
egrep '^w' data.osm.opl | egrep 'N(n[0-9]+),.*\1 '
```

Find all ways tagged with `area=yes` that are not closed:

```
egrep '^w' data.osm.opl | egrep 'area=yes' | egrep -v 'N(n[0-9]+),.*\1 '
```

Find all users who have created post boxes:

```
egrep ' v1 ' data.osm.opl | egrep 'amenity=post_box' | cut -d' ' -f7 | cut -c2- | sort -u
```

Find all node IDs used in `via` roles in relations:

```
egrep '^r' data.osm.opl | sed -e 's/^.* M\(.*\) .*$/\1/' | egrep '@via[, ]' | \
    sed -e 's/,/\n/g' | egrep '^n.*@via$' | cut -d@ -f1 | cut -c2- | sort -nu
```

Find all nodes having any tags igoring `created_by` tags:

```
egrep '^n' data.osm.opl | egrep -v ' T$' | sed -e 's/\( T\|,\)created_by=[^,]\+\(,\|$\)/\1/' | egrep -v ' T$'
```

Count tag key usage:

```
sed -e 's/^.* T//' data.osm.opl | egrep -v '^$' | sed -e 's/,/\n/g' | cut -d= -f1 | sort | uniq -c | sort -nr
```

Order by object type, object id and version (ie the usual order for OSM files):

```
sed -e 's/^r/z/' data.osm.opl | sort -b -k1.1,1.1 -k1.2,1n -k2.2,2n | sed -e 's/^z/r/'
```

Create statistics on number of nodes in ways:

```
egrep '^w' data.osm.opl | cut -d' ' -f9 | tr -dc 'n\n' | \
    awk '{a[length]++} END {for(i=1;i<=2000;++i) { print i, a[i] ? a[i] : 0 } }'
```

6.5 Reading and Writing OSM Files with Osmium

6.5.1 The osmium::io::File class

Before reading from or writing to an OSM file, you have to instantiate an object of class osmium::io::File. It encapsulates the file name as well as any information about the format of the file. In the simplest case the File class can derive the file format from the file name:

```
osmium::io::File input_file("planet.osm.pbf") // PBF format
osmium::io::File input_file("planet.osm.bz2") // XML with bzip2 compression
osmium::io::File input_file("planet.osc.gz")  // XML change file, gzip2 compression
```

The constructor of the File class has a second, optional argument giving the format of the file, which can be used if the format can't be deduced from the file name. In the simplest form the format argument looks the same as the usual file suffixes:

```
osmium::io::File input_file("somefile", "osm.bz2");
```

This setting of the format is often needed when reading from STDIN or writing to STDOUT. Both an empty string and a single dash as filename signify STDIN/STDOUT:

```
osmium::io::File input_file("-", "osm.bz2");
osmium::io::File output_file("", "pbf");
```

The format string can also take optional arguments separated by commas.

```
osmium::io::File output_file("out.osm.pbf", "pbf,pbf_dense_nodes=false");
```

Here is a list of optional arguments:

Format	Option	Default	Description
PBF	pbf_dense_nodes	true	Use DenseNodes (more space efficient)
PBF	pbf_compression	gzip	Compress blocks using gzip (use “none” to disable)
PBF	pbf_add_metadata	true	Add metadata (version, timestamp, etc. to objects)
XML	xml_change_format	false	Set change format, can also be set by using `osc` instead of `osm` suffix
XML	force_visible_flag	false	Write out `visible` flag on each object, also set if `osh` instead of `osm` suffix used

It is also possible to change the format after creating a File object using the accessor functions:

```
osmium::io::File input_file("some_file.osm");
input_file.format(osmium::io::file_format_pbf);
```

6.5.2 Reading a File

After you have a File object you can instantiate a Reader object to open the file for reading:

```
osmium::io::File input_file("input.osm.pbf");
osmium::io::Reader reader(input_file);
```

As a shortcut you can just give a file name to the Reader if you are relying on the automatic file format detection and don’t want to do any special format handling:

```
osmium::io::Reader reader("input.osm.pbf");
```

Optionally you can add a second argument to the Reader constructor giving the types of OSM entities you are interested in. Sometimes you only need, say, the ways from the file, but not the nodes and relations. If you tell the Reader about it, it might be able to read the file more efficiently by skipping those parts you are not interested in:

```
osmium::io::Reader reader("input.osm.pbf", osmium::osm_entity_bits::way);
```

You can set the following flags:

Flag	Description
`osmium::osm_entity_bits::nothing`	Do not ready any entities at all (useful if you are only interested in the file header)
`osmium::osm_entity_bits::node`	Read nodes
`osmium::osm_entity_bits::way`	Read ways
`osmium::osm_entity_bits::relation`	Read relations
`osmium::osm_entity_bits::changeset`	Read changesets
`osmium::osm_entity_bits::all`	Read all of the above

You can also “or” several flags together if needed.

You can get the header information from the file using the `header()` function:

```
osmium::io::Header header = reader.header();
```

You read the OSM entities from the file using the `read()` which returns a buffer with the data:

```
while (osmium::memory::Buffer buffer = reader.read()) {
    ...
}
```

At the end of the file an invalid buffer is returned which evaluates to false in boolean context.

You can close the file at any time. It will also be automatically closed when the Reader object goes out of scope.

```
reader.close();
```

In most cases you do not want to work with the buffers, but with the OSM entities within them. See the Iterators chapter and the Visitors and Handlers chapter for more convenient methods of working with open files.

6.5.3 The Header

Format	Option	Default	Description
all	generator	Osmium/VERSION	
XML	xml_josm_upload	not set	Set `upload` attribute in header to given value (`true` or `false`) for use in JOS

7 Iterators

Every C++ programmer is familiar with iterators and their flexibility. There is no reason we couldn't take advantage of that and of the many algorithms supplied by the STL. So libosmium supports several different kinds of iterators to access OSM data. You can iterate over all OSM objects in a buffer, or over all objects from a data source (usually a file), or over a bunch of pointers to OSM objects, and there are output iterators to write to files, too. All these different iterators can be used consistently and easily from your code without having to know much about what's underneath. And because they work just like STL iterators do, you can use all the algorithms from the STL.

Some of these iterators will keep track of underlying buffers and make sure the buffers and the data in them stay around as long as there is an iterator pointing to it. This adds some overhead but makes using the data much easier.

7.1 Accessing Data in Buffers

Buffers containing OSM entities support the usual `begin()`, `end()`, `cbegin()`, and `cend()` functions:

```
osmium::memory::Buffer buffer = ...;

auto it = buffer.begin();
auto end = buffer.end();

for (; it != end; ++it) {
    std::cout << it->type() << "\n";
}
```

Of course you can also use the C++11 `for` loop:

```
for (auto& item : buffer) {
    ...
}
```

7.2 Accessing Data from Files

```
osmium::io::Reader reader("input.osm");
osmium::io::InputIterator<osmium::io::Reader> in(reader);
osmium::io::InputIterator<osmium::io::Reader> end;
```

8 Visitors and Handlers

9 Indexes

Osmium is built around the idea that a lot of the things you want to do with OSM data can be done one OSM object at the time without having all (or large parts of) the OSM data in memory or in some kind of database. But there are many things you can not do this way. You do need some kind of storage to hold the data and some indexes to access it efficiently. Osmium provides several class templates that implement several different types of indexes.

9.1 Index Types

Osmium provides indexes modelled after the STL map and multimap classes, respectively. These classes are to be found in the osmium::index::map and osmium::index::multimap namespaces.

9.1.1 Map Index

Often we need some small, fixed amount of data stored for each OSM object. Read and write access is by ID only. Typical use cases include…

- storage of node locations where for each node ID we store the longitude and latitude of that node.
- storing the offset of an OSM object in a buffer.
- a lookup table that gives you for each node ID all IDs of the way (or ways) that include this node.

9.1.2 Storage types

There are different strategies of storing this data efficiently and there are several sub-classes of the Map and Multimap classes that use different strategies. It is important that you understand the differences and use the class thats most appropriate for your case.

The differences can be understood along different axes:

First, the question is whether the ID space is dense or not. If you are using the full planet data or large portions (such as entire continents) thereof, your ID space is dense, ie most of the possible IDs are actually present in the index. If you are only using small extracts (even with whole countries in them), you ID space is sparse, ie most of the possible IDs are not present in the index. For dense indexes data is often best stored in a kind of array indexed by the ID. For sparse indexes there are several other possibilities.

The second question is whether you have enought RAM to hold all the data in the index. Of course it is more efficient to keep the index in RAM, but if you don't have enough you need to use a disk-based index.

`handler-example.cpp`

9.1.2.1 List of map index classes see also: table in spreadsheet dummy based on vector: with stl vector, with mmap_anon, with mmap_file other: stl map, google sparsetable

9.1.2.2 List of multimap index classes based on vector: with stl vector, with mmap_anon, with mmap_file other: stl multimap, hybrid

10 Exceptions

Libosmium uses various C++ standard exceptions and some Osmium-specific exceptions to tell you about problems. All Osmium-specific exceptions are in the `osmium` namespace, they are all derived from one of the standard C++ exceptions, usually `std::runtime_error` or `std::system_error`.

10.1 List of Osmium Exceptions

`osmium::io_error` Some kind of input/output error. Derived classes describe the error in more detail.

`osmium::xml_error` (derived from `io_error`) Some kind of XML parser error.

`osmium::format_version_error` (derived from `io_error`) The OSM file format version was not understood. Osmium currently can only read version 0.6 files.

`osmium::geometry_error` Some kind of geometry error.

`osmium::projection_error` Thrown when a projection from one coordinate system into another fails in some way. Either the projection can't be initialized because of invalid parameters or the projection can't be calculated because the coordinates can't be transformed into the target coordinate system.

`osmium::not_found` This exception is thrown when a key is not found in an index.

`osmium::invalid_location`

`osmium::unknown_type` Thrown by visitors when they encounter an unknown (or in this context unexpected) item type in a buffer. This should not happen in usual circumstances.

10.2 Standard Exceptions thrown by Osmium

`std::invalid_argument` Thrown by some Osmium functions.