libgetar Documentation

Release 1.1.3

Matthew Spellings

Jun 27, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

libgetar is a library to read and write GEneric Trajectory ARchives, a binary data format designed for efficient, extensible storage of trajectory data.

Contents:

1.1 Installation and Basic Usage

1.1.1 Installation

Note: When building the *gtar* module (even when installing from PyPI), a working compiler chain is required. This means that, on Mac OSX systems, you will need to install the XCode command line tools if they are not already installed.

Versioned releases

Install a released version from PyPI using *pip*:

```
pip install gtar
```

From source

Installing the libgetar python module (`gtar`) from source is straightforward. From the root directory of the project:

```
pip install .
# Test installation:
cd
python -c 'import gtar'
```

Note that trying to run scripts from the libgetar source directory will not work!

1.1.2 Documentation

Documentation is built using sphinx and doxygen. To build it, use the Makefile in the doc subdirectory:

```
cd doc
make html
```

Note that we use the `breathe` python package to interface between sphinx and doxygen. If it isn't installed (and isn't available in your package manager), you can install it easily with pip:

```
pip install breathe --user
```

The latest version of the documentation is available online on ReadTheDocs.

1.2 The GETAR file format: GEneric Trajectory ARchives

1.2.1 The GETAR Format

The main idea behind GETAR (GEneric Trajectory ARchives; pronounced like the instrument!) files is to simply use standard data archival tools and embed a small amount of metadata into the filename within the archive. To efficiently support large trajectories, the underlying archive format (.tar.gz, .zip, .7z. . .) would ideally support efficient random access to files and should be relatively common so that generic tools would also be useful. Each backend format has different performance and stability characteristics; these are discussed in more detail below.

1.2.2 GETAR Archives

GETAR files are just normal archive files; a list of supported backends is in the *backends section*. Data are stored in individual **records**, which are simply files in the archive with a particular naming scheme.

1.2.3 Records

GETAR records consist of three primary pieces of information:

- A **name**, which indicates what attributes the data are intended to represent
- A **resolution**, which indicates at what level of detail the data are stored
- A **behavior**, which indicates how the data are stored over time

Record Names

Record names have no restriction or meaning beyond the overlying schema (properties named "position" indicate the position of particles, etc.).

Record Resolutions

There are three values that resolutions can have, two for binary data and one for text-based data:

- A **uniform** resolution indicates that the data are binary and that they represent the entire system.
- An **individual** resolution indicates that the data are binary and that they represent individual particles, rigid bodies, types…
- A **text** resolution indicates that the data are stored in plaintext and represent the entire system.

Record Behaviors

Record behaviors indicate how data are stored over time. There are three defined behaviors for different use cases:

- **Discretely varying** - Data are associated with particular times; for example, particle positions in a MD simulation
- **Continuously varying** - Data are generated "asynchronously" with the simulation; for example, the text printed to standard output during a HOOMD simulation
- **Constant** - data are stored only once and do not vary over the simulation

1.2.4 Archive Storage Paths

Put briefly, the record behavior indicates the directory within the zip archive where the data are stored and the record name and resolution are encoded in the filename. In the following paths, `{name}` will correspond to the record name, `{suffix}` will encode the storage mode of the data (determined by the resolution and binary type of the data), and `{index}` is a string which will be discussed further as needed.

Record filename suffixes are generated based on the resolution and binary type of the data stored. They are of the form `{type}.{res}`, where res is "uni" for uniform properties, "ind" for individual properties, and anything else for text properties. The type field indicates the binary storage mode of the data and is of the form `{chartype}{bitsize}`. Valid values of `{chartype}` are "i" for signed integers, "u" for unsigned integers, and "f" for floating point data. The `{bitsize}` field is the size of each element in the array of data in bits. For example, particle positions stored as 32-bit floating point numbers would be named `position.f32.ind`. Arbitrary blobs of binary data could be stored as bytestrings as `blob.u8.uni` while a JSON-encoded parameters list could be stored simply as `params.json`.

Discretely varying data are stored in `frames/{index}/{name}.{suffix}`, where the index is some meaningful string corresponding to the time of the data. Continuously varying data are stored in `vars/{name}.{suffix}/{index}`, where the index is the string representation of a natural number. Continuously varying quantities must have indices which are sequentially ordered beginning at 0 and are intended to be concatenated for use by readers. Constant quantities are stored in `{name}.{suffix}` .

Additionally, a prefix can be prepended to paths to differentiate records. For example, it could be desirable to store the moment of inertia of both individual particles as well as rigid bodies within a system. In this case, particle moments could be stored in `moment_inertia.f32.ind`, while rigid body moments could be stored in `rigid_body/moment_inertia.f32.ind` .

1.3 Supported libgetar Backends

1.3.1 Zip

The zip backend uses zip64 archives (see *Zip vs Zip64*) to store data, optionally compressed using the deflate algorithm. The zip format consists of a series of "local headers" followed by content, with a central directory at the very end of the file which lists the locations of all files present in the archive to allow for efficient random acces. This makes it possible for forcefully-killed processes to leave zip files without a central index; see *Zip Central Directories*.

Performance-wise, the zip format reads, writes, and opens files at a not-unbearably-slow rate. Its main drawback is the reliance on the presence of the central directory.

1.3.2 Tar

The tar backend stores data in the standard tar format, currently with no option of compression. The tar format stores a file header just before the data of each file, but with no global index in the standard format. Libgetar builds a global index upon opening a tar file, which consists of scanning through the entire archive file by file. Tar files should be robust to process death; in the worst case, only part of the data of a file is written.

The tar format involves the least overhead of any libgetar backend, so it is fast to read and write. However, building the index quickly becomes time-consuming for large archives with many files stored inside, causing file opens to be slow.

1.3.3 Sqlite

The sqlite backend stores data in an sqlite database. Currently, each write is implemented as a transaction, which causes the write speed to be low for large numbers of records (see the sqlite faq). Data are stored uncompressed or compressed with LZ4 and LZ4HC. Unfortunately, storing data in sqlite breaks the ability to use common archive tools to inspect and manipulate stored data, so these are less portable outside of libgetar. Because transactions are atomic, sqlite databases are robust to process death.

The sqlite backend should be expected to have moderately fast open speeds, slow write speeds (for large numbers of independent writes; use a C++ `BulkWriter` object to write multiple records within a single transaction), and fast read speeds.

1.3.4 Directory

The **experimental** directory backend stores data directly on the filesystem. Currently, data are only stored uncompressed. Because each file access occurs in the filesystem, this backend is extremely robust to process death.

1.3.5 Backend Summary

In summary:

- Zip
  - Pros
    - Reasonably fast at everything
    - "Good" compression ratio
  - Cons
    - Weak to process death

- Tar
  - Pros
    - Fast reads and writes
    - Resilient
  - Cons
    - Slow to open with many files in an archive
    - No compression
- Sqlite
  - Pros
    - Fast for reading and opening
    - Resilient
    - Fast but less-powerful compression (LZ4)
  - Cons
    - No standard archive-type tools
    - Slow for many individual writes (use `BulkWriter` for bulk writes)
- Directory
  - Pros
    - Native writing speed
    - Extremely resilient
  - Cons
    - No compression
    - Could stress filesystem with many entries

1.4 Libgetar Python Module: gtar

1.4.1 Usage

There are currently two main objects to work with in libgetar: `gtar.GTAR` archive wrappers and `gtar.Record` objects.

GTAR Objects

These wrap input and output to the zip file format and some minor serialization and deserialization abilities.

class `gtar.GTAR`

Python wrapper for the `GTAR` c++ class. Provides basic access to its methods and simple methods to read and write files within archives.

The backend is automatically selected based on the suffix of the given path: if the name ends in '.tar', a tar-format archive will be created, if it ends in '.sqlite' a sqlite-format archive will be created, if it ends in '/' a directory structure (filesystem) "archive" will be created, otherwise a zip-format archive will be created.

The open mode controls how the file will be opened.

- read: The file will be opened in read-only mode
- write: A new file will be opened for writing, potentially overwriting an existing file of the same name
- append: A file will be opened for writing, adding to the end of a file if it already exists with the same name

Parameters

- **path** – Path to the file to open
- **mode** – Open mode: one of 'r', 'w', 'a'

`close`(*self*)

Close the file this object is writing to. It is safe to close a file multiple times, but impossible to read from or write to it after closing.

`framesWithRecordsNamed`(*self*, *names*, *group=None*, *group_prefix=None*)

Returns `([record(val) for val in names], [frames])` given a set of record names names. If only given a single name, returns `(record, [frames])`.

Parameters

- **names** – Iterable object yielding a set of property names
- **group** – Exact group name to select (default: do not filter by group); overrules *group_prefix*
- **group_prefix** – Prefix of group name to select (default: do not filter by group)

`getBulkWriter`(*self*)

Get a `gtar.BulkWriter` context object. These allow for more efficient writes when writing many records at once.

`getRecord`(*self*, *Record query*, *index=''*)

Returns the contents of the given base record and index.

Parameters

- **query** (`gtar.Record`) – Prototypical `gtar.Record` object describing the record to fetch
- **index** (*string*) – Index used to fetch the record (defaults to index embedded in `query`)

Note: If an index is passed into this function, it takes precedence over the index embedded in the given record.

`getRecordTypes`(*self*, *group=None*, *group_prefix=None*)

Returns a python list of all the record types (without index information) available in this archive. Optionally filters results down to records found with a particular group name, if requested.

Parameters

- **`group`** – Exact group name to select (default: do not filter by group); overrules *group_prefix*
- **`group_prefix`** – Prefix of group name to select (default: do not filter by group)

`queryFrames`(*self*, *Record target*)

Returns a python list of all indices associated with a given record available in this archive

Parameters
`target` – Prototypical *`gtar.Record`* object (the index of which is unused)

`readBytes`(*self*, *path*)

Read the contents of the given location within the archive, or return `None` if not found

Parameters
`path` – Path within the archive to write

`readPath`(*self*, *path*)

Reads the contents of a record at the given path. Returns `None` if not found. If an array is found and the property is present in `gtar.widths`, reshape into an Nxwidths[prop] array.

Parameters
`path` – Path within the archive to write

`readStr`(*self*, *path*)

Read the contents of the given path as a string or return `None` if not found.

Parameters
`path` – Path within the archive to write

`recordsNamed`(*self*, *names*, *group=None*, *group_prefix=None*)

Returns `(frame, [val[frame] for val in names])` for each frame which contains records matching each of the given names. If only given a single name, returns `(frame, val[frame])` for each found frame. If a property is present in `gtar.widths`, returns it as an Nxwidths[prop] array.

Parameters

- **`names`** – Iterable object yielding a set of property names
- **`group`** – Exact group name to select (default: do not filter by group); overrules *group_prefix*
- **`group_prefix`** – Prefix of group name to select (default: do not filter by group)

Example:

```
g = gtar.GTAR('dump.zip', 'r')

# grab single property
for (_, vel) in g.recordsNamed('velocity'):
    pass

```

(continues on next page)

(continued from previous page)

```
# grab multiple properties
for (idx, (pos, quat)) in g.recordsNamed(['position', 'orientation']):
    pass
```

`staticRecordNamed`(*self*, *name*, *group=None*, *group_prefix=None*)

Returns a static record with the given name. If the property is found in `gtar.widths`, returns it as an Nxwidths[prop] array. Optionally restricts the search to records with the given group name or group name prefix.

Parameters

- **`name`** – Name of the property to find
- **`group`** – Exact group name to select (default: do not filter by group); overrules *group_prefix*
- **`group_prefix`** – Prefix of group name to select (default: do not filter by group)

`writeArray`(*self*, *path*, *arr*, *mode=cpp.FastCompress*, *dtype=None*)

Write the given numpy array to the location within the archive, using the given compression mode. This serializes the data into the given binary data type or the same binary format that the numpy array is using.

Parameters

- **`path`** – Path within the archive to write
- **`arr`** – Array-like object
- **`mode`** – Optional compression mode (defaults to fast compression)
- **`dtype`** – Optional numpy dtype to force conversion to

Example:

```
gtar.writeArray('diameter.f32.ind', numpy.ones((N,)))
```

`writeBytes`(*self*, *path*, *contents*, *mode=cpp.FastCompress*)

Write the given contents to the location within the archive, using the given compression mode.

Parameters

- **`path`** – Path within the archive to write
- **`contents`** – Bytestring to write
- **`mode`** – Optional compression mode (defaults to fast compression)

`writePath`(*self*, *path*, *contents*, *mode=cpp.FastCompress*)

Writes the given contents to the given path, converting as necessary.

Parameters

- **`path`** – Path within the archive to write
- **`contents`** – Object which can be converted into array or string form, based on the given path
- **`mode`** – Optional compression mode (defaults to fast compression)

`writeRecord`(*self*, *Record rec*, *contents*, *mode=cpp.FastCompress*)

Writes the given contents to the path specified by the given record.

Parameters

- **rec** – *gtar.Record* object specifying the record
- **contents** – [byte]string or array-like object to write
- **mode** – Optional compression mode (defaults to fast compression)

writeStr(*self*, *path*, *contents*, *mode=cpp.FastCompress*)

Write the given string to the given path, optionally compressing with the given mode.

Parameters

- **path** – Path within the archive to write
- **contents** – String to write
- **mode** – Optional compression mode (defaults to fast compression)

Example:

```
gtar.writeStr('params.json', json.dumps(params))
```

When writing many small records at once, a *gtar.BulkWriter* object can be used.

class gtar.BulkWriter

Class for efficiently writing multiple records at a time. Works as a context manager.

Parameters

arch – *gtar.GTAR* archive object to write within

Example:

```
with gtar.GTAR('traj.sqlite', 'w') as traj, traj.getBulkWriter() as writer:
    writer.writeStr('notes.txt', 'example text')
```

writeArray(*self*, *path*, *arr*, *mode=cpp.FastCompress*, *dtype=None*)

Write the given numpy array to the location within the archive, using the given compression mode. This serializes the data into the given binary data type or the same binary format that the numpy array is using.

Parameters

- **path** – Path within the archive to write
- **arr** – Array-like object
- **mode** – Optional compression mode (defaults to fast compression)
- **dtype** – Optional numpy dtype to force conversion to

Example:

```
writer.writeArray('diameter.f32.ind', numpy.ones((N,)))
```

writeBytes(*self*, *path*, *contents*, *mode=cpp.FastCompress*)

Write the given contents to the location within the archive, using the given compression mode.

Parameters

- **path** – Path within the archive to write
- **contents** – Bytestring to write
- **mode** – Optional compression mode (defaults to fast compression)

writePath(*self*, *path*, *contents*, *mode=cpp.FastCompress*)
Writes the given contents to the given path, converting as necessary.

Parameters

- **path** – Path within the archive to write
- **contents** – Object which can be converted into array or string form, based on the given path
- **mode** – Optional compression mode (defaults to fast compression)

writeRecord(*self*, *Record rec*, *contents*, *mode=cpp.FastCompress*)
Writes the given contents to the path specified by the given record.

Parameters

- **rec** – *gtar.Record* object specifying the record
- **contents** – [byte]string or array-like object to write
- **mode** – Optional compression mode (defaults to fast compression)

writeStr(*self*, *path*, *contents*, *mode=cpp.FastCompress*)
Write the given string to the given path, optionally compressing with the given mode.

Parameters

- **path** – Path within the archive to write
- **contents** – String to write
- **mode** – Optional compression mode (defaults to fast compression)

Example:

```
writer.writeStr('params.json', json.dumps(params))
```

Creation

```
# Open a trajectory archive for reading
traj = gtar.GTAR('dump.zip', 'r')
# Open a trajectory archive for writing, overwriting any dump.zip
# in the current directory
traj = gtar.GTAR('dump.zip', 'w')
# Open a trajectory archive for appending, if you want to add
# to the file without overwriting
traj = gtar.GTAR('dump.zip', 'a')
```

Note that currently, due to a limitation in the miniz library we use, you can't append to a zip file that's not using the zip64 format, such as those generated by python's zipfile module in most cases (it only makes zip64 if it has to for file size or count constraints; I didn't see anything right off the bat to be able to force it to write in zip64). See *Zip vs Zip64* below for solutions.

Simple API

If you know the path you want to read from or store to, you can use `GTAR.readPath()` and `GTAR.writePath()`:

```
with gtar.GTAR('read.zip', 'r') as input_traj:
    props = input_traj.readPath('props.json')
    diameters = input_traj.readPath('diameter.f32.ind')

with gtar.GTAR('write.zip', 'w') as output_traj:
    output_traj.writePath('oldProps.json', props)
    output_traj.writePath('mass.f32.ind', numpy.ones_like(diameters))
```

If you just want to read or write a string or bytestring, there are methods `GTAR.readStr()`, `GTAR.writeStr()`, `GTAR.readBytes()`, and `GTAR.writeBytes()`.

If you want to grab static properties by their name, there is `GTAR.staticRecordNamed()`:

```
diameters = traj.staticRecordNamed('diameter')
```

There are two methods that can be used to quickly get per-frame data for time-varying quantities:

1. `GTAR.framesWithRecordsNamed()` is useful for "lazy" reading, because it returns the records and frame numbers which can be processed separately before actually reading data. This is especially helpful for retrieving every 100th frame of a file, for example. This is usually the most efficient way to retrieve data.

```
(velocityRecord, frames) = traj.framesWithRecordsNamed('velocity')
for frame in frames:
    velocity = traj.getRecord(velocityRecord, frame)
    kinetic_energy += 0.5*mass*numpy.sum(velocity**2)

((boxRecord, positionRecord), frames) = traj.framesWithRecordsNamed(['box', 'position'])
good_frames = filter(lambda x: int(x) % 100 == 0, frames)
for frame in good_frames:
    box = traj.getRecord(boxRecord, frame)
    position = traj.getRecord(positionRecord, frame)
    fbox = freud.box.Box(*box)
    rdf.compute(fbox, position, position)
    matplotlib.pyplot.plot(rdf.getR(), rdf.getRDF())
```

2. `GTAR.recordsNamed()`: is useful for iterating over **all** frames in the archive. It reads and returns the content of the records it finds.

```
for (frame, vel) in traj.recordsNamed('velocity'):
    kinetic_energy += 0.5*mass*numpy.sum(vel**2)

for (frame, (box, position)) in traj.recordsNamed(['box', 'position']):
    fbox = freud.box.Box(*box)
    rdf.compute(fbox, position, position)
    matplotlib.pyplot.plot(rdf.getR(), rdf.getRDF())
```

Advanced API

The more complicated API can be used if you have multiple properties with the same name (for example, a set of low-precision trajectories for visualization and a less frequent set of dumps in double precision for restart files).

Finding Available Records

A list of record types (records with blank indices) can be obtained by the following:

```
traj.getRecordTypes()
```

This can be filtered further in something like:

```
positionRecord = [rec for rec in traj.getRecordTypes() if rec.getName() == 'position'][0]
```

The list of frames associated with a given record can be accessed as:

```
frames = traj.queryFrames(rec)
```

Reading Binary Data

To read binary data (in the form of numpy arrays), use the following method:

```
traj.getRecord(query, index="")
```

This takes a *gtar.Record* object specifying the path and an optional index. Note that the index field of the record is nullified in favor of the index passed into the method itself; usage might look something like the following:

```
positionRecord = [rec for rec in traj.getRecordTypes() if rec.getName() == 'position'][0]
positionFrames = traj.queryFrames(positionRecord)
positions = [traj.getRecord(positionRecord, frame) for frame in positionFrames]
```

Record Objects

These objects are how you discover what is inside an archive and fetch or store data. Records consist of several fields defining where in the archive the data are stored, what type the data are, and so forth. Probably the most straightforward way to construct one of these yourself is to let the Record constructor itself parse a path within an archive:

```
rec = Record('frames/0/position.f32.ind')
```

`class gtar.`**`Record`**

Python wrapper for the c++ Record class. Provides basic access to Record methods. Initializes in different ways depending on the number of given parameters.

- No arguments: default constructor
- 1 argument: Parse the given path
- 6 arguments: Fill each field of the Record object (*group*, *name*, *index*, *behavior*, *format*, *resolution*)

`getBehavior`(*self*)

Returns the *behavior* field of this object

getFormat(*self*)

Returns the *format* field of this object

getGroup(*self*)

Returns the *group* field of this object

getIndex(*self*)

Returns the *index* field for this object

getName(*self*)

Returns the *name* field of this object

getPath(*self*)

Generates the path of the file inside the archive for this object

getResolution(*self*)

Returns the *resolution* field for this object

nullifyIndex(*self*)

Nullify the index field of this object

setIndex(*self*, *index*)

Sets the *index* field of this object

1.4.2 Tools

gtar.fix

Fix a getar-formatted zip file.

```
usage: python -m gtar.fix [-h] [-o OUTPUT] input

Command-line zip archive fixer

positional arguments:
  input                 Input zip file to read

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
Output location for fixed zip archive
```

gtar.cat

Take records from multiple getar-formatted files and place them into an output file. In case of name conflicts, records from the last input file take precedence.

```
usage: cat.py [-h] [-o OUTPUT] ...

Command-line archive concatenation

positional arguments:
  inputs                Input files to read

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        File to write to
```

gtar.copy

Copy each record from one getar-formatted file to another.

```
usage: python -m gtar.copy [-h] input output

Command-line archive copier or translator

positional arguments:
  input           Input file to read
  output          File to write to

optional arguments:
  -h, --help      show this help message and exit
```

gtar.read

Create an interactive python shell with the given files opened for reading.

```
usage: read.py [-h] ...

Interactive getar-format archive shell

positional arguments:
  inputs      Input files to open

optional arguments:
  -h, --help  show this help message and exit
```

1.4.3 Enums: OpenMode, CompressMode, Behavior, Format, Resolution

`class gtar.OpenMode`

Enum for ways in which an archive file can be opened

Read

Write

Append

`class gtar.CompressMode`

Enum for ways in which files within an archive can be compressed

NoCompress

FastCompress

MediumCompress

SlowCompress

`class gtar.Behavior`

Enum for how properties can behave over time

`Constant`

`Discrete`

`Continuous`

`class gtar.Format`

Formats in which binary properties can be stored

`Float32`

`Float64`

`Int32`

`Int64`

`UInt8`

`UInt32`

`UInt64`

`class gtar.Resolution`

Resolution at which properties can be recorded

`Text`

`Uniform`

`Individual`

1.5 Libgetar C++ API

1.5.1 GTAR

class **GTAR**

Accessor interface for a trajectory archive.

Public Functions

GTAR(const std::string &filename, const OpenMode mode)

Constructor. Opens the file at filename in the given mode. The format of the file depends on the extension of filename.

void **close**()

Manually close the opened archive (it automatically closes itself upon destruction)

void **writeString**(const std::string &path, const std::string &contents, CompressMode mode)

Write a string to the given location.

void **writeBytes**(const std::string &path, const std::vector<char> &contents, CompressMode mode)

Write a bytestring to the given location.

void **writePtr**(const std::string &path, const void *contents, const size_t byteLength, CompressMode mode)

Write the contents of a pointer to the given location.

template<typename **iter**, typename **T**>
void **writeIndividual**(const std::string &path, const *iter* &start, const *iter* &end, CompressMode mode)

Write an individual binary property to the specified location, converting to little endian if necessary.

template<typename **T**>
void **writeUniform**(const std::string &path, const *T* &val)

Write a uniform binary property to the specified location, converting to little endian if necessary.

template<typename **T**>
SharedArray<*T*> **readIndividual**(const std::string &path)

Read an individual binary property to the specified location, converting from little endian if necessary.

template<typename **T**>
SharedPtr<*T*> **readUniform**(const std::string &path)

Read a uniform binary property to the specified location, converting from little endian if necessary.

SharedArray<char> **readBytes**(const std::string &path)

Read a bytestring from the specified location.

std::vector<*Record*> **getRecordTypes**() const

Query all of the records in the archive. These will all have empty indices.

std::vector<std::string> **queryFrames**(const *Record* &target) const

Query the indices associated with a given record. The record is not required to have a null index.

class **BulkWriter**

Public Functions

BulkWriter(*GTAR* &archive)

Create a new *BulkWriter* on an archive. Only one should exist for any archive at a time.

~BulkWriter()

Clean up the *BulkWriter* data. Causes all writes to be performed.

void **writeString**(const std::string &path, const std::string &contents, CompressMode mode)

Write a string to the given location.

void **writeBytes**(const std::string &path, const std::vector<char> &contents, CompressMode mode)

Write a bytestring to the given location.

void **writePtr**(const std::string &path, const void *contents, const size_t byteLength, CompressMode mode)

Write the contents of a pointer to the given location.

template<typename **iter**, typename **T**>
void **writeIndividual**(const std::string &path, const *iter* &start, const *iter* &end, CompressMode mode)

Write an individual binary property to the specified location, converting to little endian if necessary.

template<typename **T**>
void **writeUniform**(const std::string &path, const *T* &val)

Write a uniform binary property to the specified location, converting to little endian if necessary.

1.5.2 Record

class **Record**

Simple class for a record which can be stored in an archive.

Public Functions

Record()

Default constructor: initialize all strings to empty, behavior to Constant, format to UInt8, and resolution to Text

Record(const std::string &path)

Create a record from a path (inside the archive), parsing the path into the various fields

Record(const std::string &group, const std::string &name, const std::string &index, *Behavior* behavior, *Format* format, *Resolution* resolution)

Create a record directly from the full set of elements.

Record(const *Record* &rhs)

Copy constructor.

void **operator=**(const *Record* &rhs)

Assignment operator.

bool **operator==**(const *Record* &rhs) const

Equality.

bool **operator!=**(const *Record* &rhs) const

Inequality.

bool **operator<**(const *Record* &rhs) const

Comparison.

void **copy**(const *Record* &rhs)

Copy all fields from rhs into this object.

std::string **`nullifyIndex`**`()`

Set our index to the empty string.

Record **`withNullifiedIndex`**`()` const

Return a copy of this object, but with an empty string for its index

std::string **`getPath`**`()` const

Construct a path (for inside an archive) from this object's various fields

std::string **`getGroup`**`()` const

Get the stored group field.

std::string **`getName`**`()` const

Get the stored name field.

std::string **`getIndex`**`()` const

Get the stored index field.

Behavior **`getBehavior`**`()` const

Get the stored behavior field.

Format **`getFormat`**`()` const

Get the stored format field.

Resolution **`getResolution`**`()` const

Get the stored resolution field.

void **`setIndex`**`(`const std::string &index`)`

Set the index field for this *Record* object.

1.5.3 Enums: Behavior, Format, Resolution

enum `gtar::`**`Behavior`**

Time behavior of properties.

Values:

enumerator **`Constant`**

enumerator **`Discrete`**

enumerator **`Continuous`**

enum `gtar::`**`Format`**

Binary formats in which properties can be stored.

Values:

enumerator **`Float32`**

enumerator **`Float64`**

enumerator **`Int32`**

enumerator **`Int64`**

enumerator **`UInt8`**

enumerator **`UInt32`**

enumerator **`UInt64`**

enum `gtar::`**`Resolution`**

Level of detail of property storage.

Values:

enumerator **`Text`**

enumerator **`Uniform`**

enumerator **`Individual`**

1.5.4 SharedArray

template<typename **`T`**>

class **`SharedArray`**

Generic reference-counting shared array implementation for arbitrary datatypes.

Subclassed by gtar::SharedPtr< T >

Public Functions

inline **`SharedArray`**`()`

Default constructor. Allocates nothing.

inline **`SharedArray`**`(`*T* *target, size_t length`)`

Target constructor: allocates a new SharedArrayShim for the given pointer and takes ownership of it.

inline **`SharedArray`**`(`const *SharedArray*<*T*> &rhs`)`

Copy constructor: make this object point to the same array as rhs, increasing the reference count if necessary

inline **`SharedArray`**`(`const SharedPtr<*T*> &rhs`)`

Initialize from SharedPtr: make this object point to the same shim as rhs, increasing the reference count if necessary

inline **`~SharedArray`**`()`

Destructor: decrement the reference count and deallocate if we are the last owner of the pointer inline void **copy**(const *SharedArray*<*T*> &rhs)

Non-operator form of assignment.

inline bool **isNull**()

Returns true if m_shim is null or m_shim's target is null.

inline void **operator=**(const *SharedArray*<*T*> &rhs)

Assignment operator: make this object point to the same thing as rhs (and deallocate our old memory if necessary)

inline iterator **begin**()

Returns a standard style iterator to the start of the array.

inline iterator **end**()

Returns a standard style iterator to just past the end of the array.

inline *T* ***get**()

Returns the raw pointer held (NULL otherwise)

inline size_t **size**() const

Returns the size, in number of objects, of this array.

inline void **release**()

Release our claim on the pointer, including decrementing the reference count

inline *T* ***disown**()

Stop managing this array and give it to C.

inline void **swap**(*SharedArray*<*T*> &target)

Swap the contents of this array with another.

inline *T* &**operator[]**(size_t idx)

Access elements by index.

inline const *T* &**operator[]**(size_t idx) const

Const access to elements by index.

1.6 Known Issues and Solutions

1.6.1 Zip Central Directories

The zip file format stores a "table of contents" known as a central directory at the end of the file. This allows zip archives to be "random-access" in the sense that you don't have to visit every file in the archive to know what files exist in the archive, but if a process is terminated forcefully (kill -9 or hitting a wall clock limit), libgetar will not get an opportunity to write the central directory. In this case, the zip file will be unreadable until you rebuild the central directory using the command line tool `zip -FF` or the python module *`gtar.fix`* (which uses `zip -FF` and deletes all data from any frames that were removed in the process). Example:

```
python -m gtar.fix broken.zip -o fixed.zip
```

Some very large (>8GB) zip files seem to be unable to be fixed, even with `zip -FF`. In this case, to recover your data you can extract it all using the `jar` tool, which does not even look at the central directory when extracting:

```
mkdir temp && cd temp
jar xvf ../broken.zip
zip -mr fixed.zip -xi ./*
```

1.6.2 Zip vs Zip64

The zip archives libgetar writes are always in the zip64 format. It can read "normal" zip archives just fine, but appending to them will not work since converting an archive in-place is unsafe in case of errors. Running the `gtar.fix` or `gtar.copy.main()` python modules will always convert a file to zip64 format. Example:

```
python -m gtar.copy 32bit.zip 64bit.zip
```

1.6.3 basic_string::_S_construct null not valid

This is due to passing in a python string object instead of a bytes object and is probably an error on my part. These errors look like this:

```
terminate called after throwing an instance of 'std::logic_error'
  what():  basic_string::_S_construct null not valid
```

If you see any of those, let me know!

1.6.4 ImportError: cannot import name '_gtar' from partially initialized module 'gtar'

This typically stems from trying to import the *gtar* source directory directly as a python module. Since the python module depends on a compiled C extension, it must first be installed separately. To fix this, install the module (see *Installation and Basic Usage*) and run python outside of the *libgetar* source directory to avoid python automatically importing the *gtar* source directory.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX