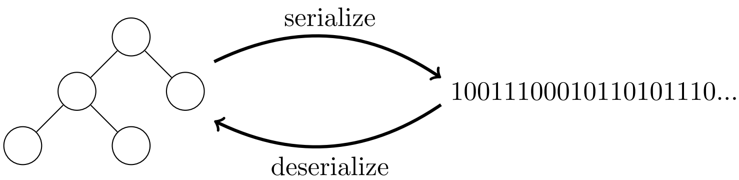
Homework Assignment 7

Due Date: 11:59, June 18, 2015

1 Background

In computer science, **serialization** is the process of transforming object states or data structures into a format that can be stored in a file (or transmitted across the network) and reconstructed (or **deserialized**) later by the same or another program. For example, we may want to serialize a data structure (or an object) in one program into a sequence of bytes and to reconstruct the data structure (or the object) from the sequence of bytes in another program, as illustrated in the following diagram.



C++ does not provide direct support for serialization. However, it is possible to write your own serialization functions since C++ supports writing binary data. Nevertheless, serialization is a complicated subject with many design issues to tackle, for example:

- How many bits and which byte order (big-endian or little-endian) are used to store an integer?
- How are fields in an object delimited?
- What other types are supported? How to properly serialize IEEE 754 floating point numbers?
- Is the format optimized for size or efficiency of (de)serialization? Is the format human-readable?

Fortunately, there have been many solutions proposed to tackle the problems. In this assignment, we are set to explore the design and implementation of binary serialization through an existing example: **MessagePack** (`http://msgpack.org/`), an efficient object serialization format designed to be compact and simple by Sadayuki Furuhashi. MessagePack consists of a *data type system* and a description of the *binary format*, as well as conversions between the two. Serialization converts application objects into the MessagePack format via MessagePack's type system, whereas deserialization converts from the MessagePack binary format into application objects. The full MessagePack specification can be found on its GitHub repository at:

`https://github.com/msgpack/msgpack/blob/master/spec.md`

In this assignment, we will only implement a subset of the MessagePack specification. While the specification below is vastly simplified in order to ease implementation, our version of MessagePack is fully compatible with original format.

2 Problem Statement

You are asked to implement a class, called `value` and defined in Section 2.3, which allows a client program to serialize certain types of objects into specific binary formats, and vice versa. Sections 2.1 and 2.2 describe the data types that the simplified MessagePack supports for serialization and the serialized formats of these data types, respectively.

2.1 Type System

A MessagePack object (i.e., an instance of the `value` class) can be a value of one of the following types: **Nil**, **Boolean**, **Integer**, **String**, **Array**, and **Map** types.

2.1.1 Nil

A MessagePack object of type **Nil** may have only one value: **nil**.

2.1.2 Boolean

A MessagePack object of type **Boolean** may have two possible values: **true** and **false**.

2.1.3 Integer

A MessagePack object of type **Integer** may contain any integral values within a signed 64-bit integer: -2^{63} to $2^{63} - 1$.

2.1.4 String

A MessagePack object of type **String** represents an ASCII string up to $2^{32} - 1$ characters in length.

2.1.5 Array

A MessagePack object of type **Array** represents an ordered sequence of MessagePack objects up to $2^{32} - 1$ elements in size. We denote such an array as its values seperated by commas and then enclosed in square brackets, for example: [**1, 2**], [**“foo”, “bar”**], [**“foo”, true**], [**-1,** [**“nested”**]**, nil**], [].

Note that **Array**s may be nested, as shown in the fourth example.

2.1.6 Map

A MessagePack object of type **Map** represents an associative map from **String** to MessagePack objects. Keys in a **Map** must be unique, and a **Map** may contain up to $2^{32} - 1$ key-value pairs in size. We denote a map by seperating each key-value pair by commas enclosed in curly braces, for example: **{“a”: 1, “b”: true}**, **{“array”:** [**1, true**]**}**, **{}**.

Like arrays, **Map**s may also be nested. Furthermore, both **Array**s and **Map**s may also be arbitrarily nested, e.g. [**{“a”:** [**{ }**]**}**].

2.2 Serialization Format

For each type of MessagePack objects, we define a serialization format of its value. Each serialized object begins with an octet which acts as a *tag* for the type of the object. The tag is followed by zero or more bytes that further describe the value of the object.

2.2.1 Nil

A value **nil** is represented as a single octet: **0xc0**.

2.2.2 Boolean

A **Boolean** is serialized into a single octet: **0xc2** if the value is **false**, or **0xc3** if it is **true**.

2.2.3 Integer

An **Integer** is serialized into 9 octets: the tag for the **Integer** type is defined as **0xd3**. The tag is then followed by 8 octets that represents the value as a 64-bit big-endian signed integer.

For example, the integer **123** is serialized into:

d3	00	00	00	00	00	00	00	7b
tag	123							

The integer **-123** is serialized into:

d3	ff	ff	ff	ff	ff	ff	ff	85
tag	-123							

2.2.4 String

An object of type **String** is represented by a series of octets. The tag is defined to be **0xdb**, which is followed by the length of the **String** stored as a **32-bit big-endian unsigned** integer. Then, the actual content of the **String** follows byte-by-byte. Note that the terminating NUL character (`'\0'`) in C-style strings should not be stored nor included in the calculation of length.

For instance, the **String "msgpack"** should be serialized into:

db	00	00	00	07	6d	73	67	70	61	63	6b
tag	length = 7				'm'	's'	'g'	'p'	'a'	'c'	'k'

The empty **String ""** should be simply serialized as:

db	00	00	00	00
tag	length = 0			

2.2.5 Array

An **Array** is encoded as **0xdd**, followed by the size of the array stored as a **32-bit big-endian unsigned** integer and finally the serialized result of each value in the array.

For example, the serialized result of the **Array [1, true]** is:

dd	00	00	00	02	d3	00	00	00	00	00	00	00	01	c3
tag	size = 2				1									true

The nested **Array** [[]] is encoded as:

dd	00	00	00	01	dd	00	00	00	00
tag	size = 1				[]				

2.2.6 Map

A **Map** is serialized as **0xdf**, followed by the size of the map stored as a **32-bit big-endian unsigned** integer and finally the serialized result of each key-value pair.

For example, the **Map {"a": true, "b": {}}** is serialized as:

df	00	00	00	02	db 00 00 00 01 61	c3	db 00 00 00 01 62	df	00	00	00	00
tag	size = 2				"a"	true	"b"	{ }				

2.3 The `value` class

A MessagePack object is an instance of the `value` class in C++. A skelton of the class and its required interfaces is already provided for you in the file `msgpack.hpp`.

2.3.1 Public members

```
typedef bool                          boolean_type;
typedef std::int64_t                  integer_type;
typedef std::string                   string_type;
typedef std::vector<value>            array_type;
typedef std::map<string_type, value> map_type;
```

Define the C++ types used to represent each kind of MessagePack object. Note that `std::int64_t` is defined to be an integral type with exactly 64 bits in the header `<cstdint>`. The type for **Nil** is also omitted as we never have to store such a value.

1. ```
   value();
   explicit value(const boolean_type);
   explicit value(const integer_type);
   explicit value(const string_type&);
   explicit value(const array_type&);
   explicit value(const map_type&);
   ```
   Construct a `value` of the corresponding type of object.

2. ```
   value(const value&);
   value& operator=(const value&);
   ~value();
   ```
   A `value` should be copy-constructable, copy-assignable, and destructable.

3. ```
   bool is_nil() const;
   bool is_boolean() const;
   bool is_integer() const;
   bool is_string() const;
   bool is_array() const;
   bool is_map() const;
   ```
   Return whether the stored object in a `value` is the type queried.

4. ```
   boolean_type& get_boolean();
   integer_type& get_integer();
   string_type&  get_string();
   array_type&   get_array();
   map_type&     get_map();
   ```
   Return the stored object of the specified type. If the `value` does not currently store an object of the type requested, throw a `std::bad_cast` exception.

5. `const boolean_type& get_boolean() const;`
   `const integer_type& get_integer() const;`
   `const string_type&  get_string()  const;`
   `const array_type&   get_array()   const;`
   `const map_type&     get_map()     const;`
   The `const` overloads of (4). Return a const reference to stored object or throw a `std::bad_cast` exception.

6. `friend bool operator==(const value& lhs, const value& rhs);`
   `friend bool operator!=(const value& lhs, const value& rhs);`
   `value`s are comparable. Two `value`s are considered equal only if they store the same type of object and the stored objects are equal. Note that two `nil`s are considered to be equal.

7. `std::ostream& serialize(std::ostream& out) const;`
   Write the serialized result of a `value` into the `std::ostream` and return `out`.

8. `static value deserialize(std::istream& in);`
   Read and construct a `value` from a `std::istream`. If the input is invalid, return a `value` representing **nil** and set the failbit on `in`.

2.3.2 Implementation Hints

A `value` is able to represent multiple types of objects, but only one of them may be active at one time. In order to save space, a *tagged union* is used to store the object internally. A tagged union consists of a tag and a union. The tag is used to record the current type that is active in the union, and is usually implemented as an enumeration (`enum`). The union is a `union` of all possible types that the tagged union may store. For example, a tagged union of `int` and `std::string` may be declared as:

```
using std::string;

enum class tag_t { int_tag, string_tag };
union union_t {
    // The ctors. and dtor. must be explicitly defined.
    //
    // Their bodies perform no operations as it is expected that
    // the caller will be responsible for constructing and destructing
    // the members correctly.
    union_t() {}
    ~union_t() {}

    int i;
    string s;
};

tag_t t;
union_t u;
```

The tag `t` here is used to mark the current active type of `u`. For example, to construct a tagged union of `int` 0, we set the tag `t` to `tag_t::int_tag` and use the *placement new* syntax to construct an `int` at memory location `&u.i`:

```
t = tag_t::int_tag;
new (&u.i) int(0);
// use u.i
```

To store a `std::string` instead, we re-assign the tag to `tag_t::string_tag` and construct a `std::string` at `&u.s`:

```
t = tag_t::string_tag;
new (&u.s) string("foo");
// use u.s
```

Note that, suppose we want to store an `int` again, we must destruct the `std::string` first by calling the destructor explicitly in order to avoid memory leaks:

```
u.s.~string();
t = tag_t::int_tag;
new (&u.i) int(2);
// use u.i
```

Care must be taken to ensure that the tag always corresponds to the active type. For instance, the following code will produce a runtime error:

```
t = tag_t::string_tag; // Wrong tag!
new (&u.i) int(3);

switch (t) {
   case tag_t::int_tag:
      std::cout << u.i;
      break;
   case tag_t::string_tag:
      std::cout << u.s; // Error: accessing u.s which is inactive right now!
      break;
}
```

We may now define the internal structure of `value` as the following:

```
enum class tag_t { nil_tag, boolean_tag, integer_tag,
                   string_tag, array_tag, map_tag };
union union_t {
    union_t() {}
    ~union_t() {}

    boolean_type b;
    integer_type i;
    string_type s;
    array_type a;
    map_type m;
};

tag_t tag;
union_t val;
```

Your `Value` constructors should be responsible maintaining the relationship between `tag` and `val` members, and your `Value` destructor should destruct the object stored in the union if needed. Remember that you should check the `tag` variable before attempting to access objects in the union, and report an error if the requested type of object is not active.

3 Examples

Construct a `value` of **Integer 1** and verify that we can get the stored integer back:

```
const value v1(std::int64_t(1));
assert(v1.is_integer());
assert(v1.get_integer() == 1);
```

Consturct another `value` of **String "msgpack"**:

```
value v2(std::string("msgpack"));
assert(v2.is_string());
```

Attempting to get a **Boolean** from `v2` should fail and throw an exception:

```
try {
    v2.get_boolean();
    // should not reach here
} catch (const std::bad_cast&) {}
```

Note that we can also modify the `value` via the reference returned by `get_string()`:

```
v2.get_string() = "hello";
```

Now, we construct yet another `value` of **Array** type — [**true, "hello"**]:

```
std::vector<value> a;
a.push_back(value(true));
a.push_back(v2);
const value v3(a);
assert(v3.is_array());
```

Serialize `v3` to the file `v3.bin`:

```
std::ofstream out("v3.bin");
v3.serialize(out);
```

Verify that the content of `v3.bin` matches the MessagePack format:

```
$ hexdump -C v3.bin
00000000  dd 00 00 00 02 c3 db 00  00 00 05 68 65 6c 6c 6f  |..........hello|
```

Finally, we deserialize the content of `v3.bin` back to a `value` and verify that it is the same as `v3`:

```
std::ifstream in("v3.bin");
const value v4 = value::deserialize(in);
assert(v3 == v4);
```

See the included `main.cpp` file for a more comprehensive test of all required functionalities in this assignment.

4 Hints and Notes

- To perform conversion from your machine's host byte order to big-endian, you may use the `htobe*()` (**h**ost byte order **to** **b**ig-**e**ndian) functions in the provided header file `"endian.h"`. The file provides standardized endianness conversion routines on Windows, Linux, Mac OS X and other UNIX-like operating systems. (Credit: `https://gist.github.com/panzi/6856583`, released into public domain)
- Use a hex editor to examine and debug your serialization result. On UNIX-like systems, you may simply invoke the **`hexdump`** program (See the examples section for a simple usage). Alternatively, if you prefer to use a GUI program, Bless (`http://home.gna.org/bless/index.html`) is an open source and cross-platform hex editor that supports both Windows and GNU/Linux.
- Your classes and functions must correctly manage all allocated memory under any circumstances. There should be no memory leaks, dangling pointers or uses of uninitialized memory within your implementation. If the presence of memory-related bugs cause the test program to crash, you will receive **zero** credits for the test case.
- Test all edge cases of your implementation, for example, malformed input during deserialization should not cause your program to crash. You should also test your implementation with multiple compilers and optimization settings, which may help revealing bugs within your program.
- Print your own debug messages to standard error by using `std::cerr`. TAs will ignore all outputs on the standard error stream.
- (Optional) If you are feeling adventurous, you are allowed and recommended to use `boost::variant` as a replacement of the tagged union described in section 2.3.2. `boost::variant` is a type-safe implementation of the tagged union concept which lifts the burden of correctly maintaining the tag and the union from the programmer. As of 1.58.0, Boost also includes the Boost.Endian library that provides endian-specific (e.g. `boost::endian::big_uint32_t`) types and conversion routines. For more information and documentation, see `http://www.boost.org/`. Boost 1.58 will be available on TA's workstation when compiling your program.

5 Submission

Archive your homework into a single zip file and submit the zipped file to E3. The zipped file must be named by your student ID (e.g. `0123456.zip`) and **must** contain the following file(s) **in the root directory**:

- `msgpack.hpp`: Your implementation of the `value` class and (de)serialization routines.

You are allowed to submit and include other header files you have written. TAs will compile your homework using `clang-3.6` with the following command on a Linux workstation:

```
$ clang++ -std=c++14 -Wall -Wextra -pedantic -O2 main.cpp
```

where `main.cpp` will `#include` your `msgpack.hpp` in order to test your implementation.

Verify that your program can be successfully compiled and run without problems on the aforementioned environment before submitting. Contact TAs for assistance if you have problems testing your program.