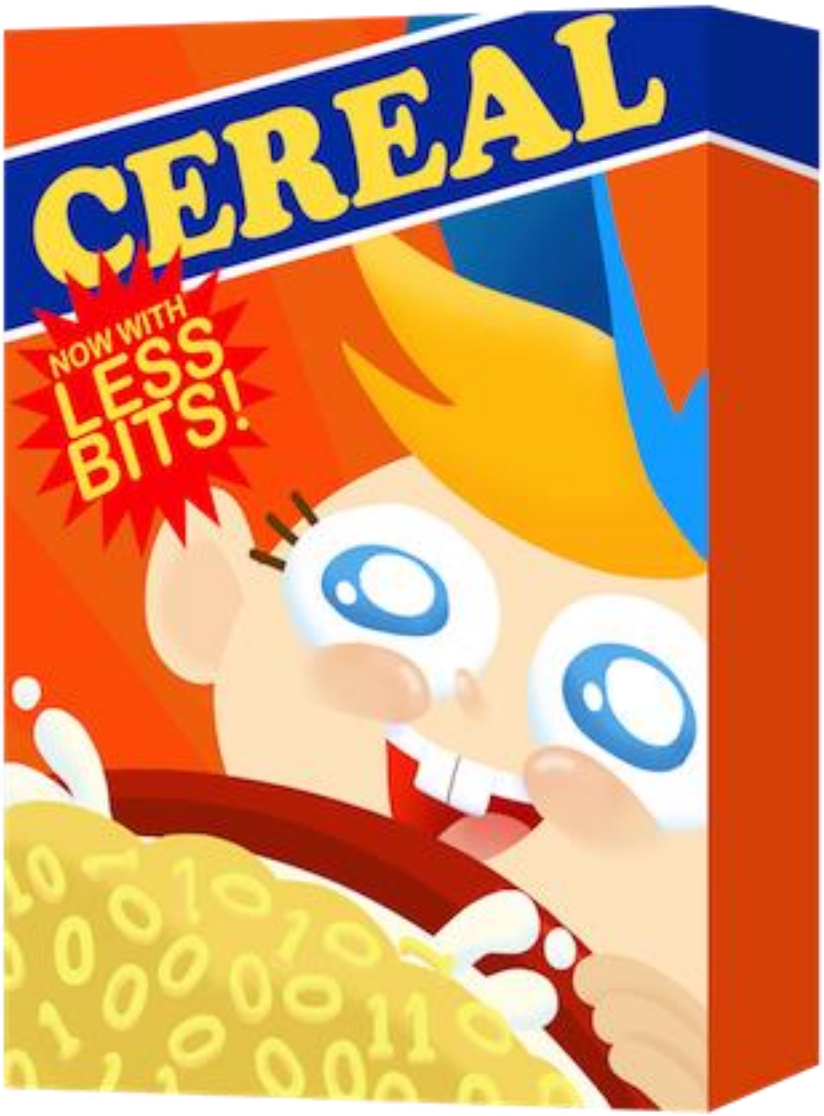
cereal - A C++11 library for serialization

https://github.com/USCiLab/cereal



cereal is a header-only C++11 serialization library. cereal takes arbitrary data types and reversibly turns them into different representations, such as compact binary encodings, XML, or JSON. cereal was designed to be fast, light-weight, and easy to extend - it has no external dependencies and can be easily bundled with other code or used standalone.

cereal has great documentation

Looking for more information on how cereal works and its documentation?
Visit cereal's web page to get the latest information.

cereal is easy to use

Installation and use of of cereal is fully documented on the main web page, but this is a quick and dirty version:

- Download cereal and place the headers somewhere your code can see them
- Write serialization functions for your custom types or use the built in support for the standard library cereal provides
- Use the serialization archives to load and save data

```
#include <cereal/types/unordered_map.hpp>
#include <cereal/types/memory.hpp>
#include <cereal/archives/binary.hpp>
#include <fstream>

struct MyRecord
```

```
{
  uint8_t x, y;
  float z;

  template <class Archive>
  void serialize( Archive & ar )
  {
    ar( x, y, z );
  }
};

struct SomeData
{
  int32_t id;
  std::shared_ptr<std::unordered_map<uint32_t, MyRecord>> data;

  template <class Archive>
  void save( Archive & ar ) const
  {
    ar( data );
  }

  template <class Archive>
  void load( Archive & ar )
  {
    static int32_t idGen = 0;
    id = idGen++;
    ar( data );
  }
};

int main()
{
  std::ofstream os("out.cereal", std::ios::binary);
  cereal::BinaryOutputArchive archive( os );

  SomeData myData;
  archive( myData );

  return 0;
}
```

cereal has a mailing list

Either get in touch over email or on the web.

Quick Start

This is a quick guide to get **cereal** up and running in a matter of minutes. The only prerequisite for running **cereal** is a modern C++11 compliant compiler, such as GCC 4.7.3, clang 3.3, MSVC 2013, or newer. Older versions might work, but we can't guarantee it.

Get cereal

cereal can be directly included in your project or installed anywhere you can access header files. Grab the latest version from Github or use the download links above, then drop the **cereal** folder from the include directory (`cereal_base_dir/include/cereal`) somewhere your project can find. There's nothing to build or make - **cereal** is header only.

Add serialization methods for your classes

cereal needs to know which data members to serialize in your classes. Let it know by implementing a `serialize` method in your class:

Explain

```
struct MyClass
{
  int x, y, z;

  // This method lets cereal know which data members to serialize
  template<class Archive>
  void serialize(Archive & archive)
  {
    archive( x, y, z ); // serialize things by passing them to the
archive
  }
};
```

cereal also offers more flexible ways of writing serialization functions such as moving them outside of class definitions or splitting them into separate load and save functions. You can read all about that in the serialization functions section of the

documentation. **cereal** can also support class versioning, private serialization methods, and even classes that don't support default construction.

You can serialize primitive data types and nearly every type in the standard library without needing to write anything yourself.

Choose an archive

cereal currently supports three basic archive types: binary (also available in a portable version), XML, and JSON. These archives are the middlemen between your code and your serialized data - they handle the reading and writing for you. XML and JSON archives are human readable but lack the performance (both space and time) of the binary archive. You can read all about these archives in the archives section of the documentation.

Include your preferred archive with one of:

- `#include <cereal/archives/binary.hpp>`
- `#include <cereal/archives/portable_binary.hpp>`
- `#include <cereal/archives/xml.hpp>`
- `#include <cereal/archives/json.hpp>`

Serialize your data

Create a **cereal** archive and send the data you want to serialize to it. Archives are designed to be used in an RAII manner and are guaranteed to flush their contents only on destruction (though it may happen earlier). Archives generally take either an `std::istream` or an `std::ostream` object in their constructor:

Explain

```
#include <cereal/archives/binary.hpp>
#include <sstream>

int main()
{
  std::stringstream ss; // any stream can be used

  {
```

```
    cereal::BinaryOutputArchive oarchive(ss); // Create an output
archive

    MyData m1, m2, m3;
    oarchive(m1, m2, m3); // Write the data to the archive
   } // archive goes out of scope, ensuring all contents are flushed

   {
    cereal::BinaryInputArchive iarchive(ss); // Create an input archive

    MyData m1, m2, m3;
    iarchive(m1, m2, m3); // Read the data from the archive
   }
  }
```

Important! If you didn't read that paragraph about **cereal** using RAII, read it again! Some archives in **cereal** can only safely finish flushing their contents upon their destruction. Make sure, especially for output serialization, that your archive is automatically destroyed when you are finished with it.

Naming values

cereal also supports name-value pairs, which lets you attach names to the objects it serializes. This is only truly useful if you choose to use a human readable archive format such as XML or JSON:

Explain

```
#include <cereal/archives/xml.hpp>
#include <fstream>

int main()
{
  {
    std::ofstream os("data.xml");
    cereal::XMLOutputArchive archive(os);

    MyData m1;
    int someInt;
    double d;

    archive( CEREAL_NVP(m1), // Names the output the same as the
variable name
             someInt,      // No NVP - cereal will automatically generate
an enumerated name
```

```
            cereal::make_nvp("this_name_is_way_better", d) ); // specify a
name of your choosing
    }

    {
      std::ifstream is("data.xml");
      cereal::XMLInputArchive archive(is);

      MyData m1;
      int someInt;
      double d;

      archive( m1, someInt, d ); // NVPs not strictly necessary when
loading
                           // but could be used (even out of order)
    }
  }
```

More information about name-value pairs can be found by reading the relevent entries in the doxygen documentation on utility functions.

Learn more

cereal can do much more than these simple examples demonstrate. **cereal** can handle smart pointers, polymorphism, inheritance, and more. Take a tour of its features by following the documentation or diving into the doxygen documentation.

Standard Library Support

cereal supports most of the containers and classes found in the C++ standard library out of the box.

TLDR Version

cereal supports pretty much everything in the C++ standard library. Include the proper header from **cereal** to enable support (e.g. `<cereal/types/vector.hpp>`. See the doxygen docs for a complete list of supported types.

STL Support

To use a type found in the standard library, just include the proper header from `#include <cereal/types/xxxx.hpp>` and serialize data as you normally would:

Explain

```
// type support
#include <cereal/types/map.hpp>
#include <cereal/types/vector.hpp>
#include <cereal/types/string.hpp>
#include <cereal/types/complex.hpp>

// for doing the actual serialization
#include <cereal/archives/json.hpp>
#include <iostream>

class Stuff
{
 public:
  Stuff() = default;

  void fillData()
  {
   data = { {"real", { {1.0f, 0},
           {2.2f, 0},
           {3.3f, 0} }},
```

```
      {"imaginary", { {0, -1.0f},
            {0, -2.9932f},
            {0, -3.5f} }} };
  }

 private:
  std::map<std::string, std::vector<std::complex<float>>> data;

  friend class cereal::access;

  template <class Archive>
  void serialize( Archive & ar )
  {
   ar( CEREAL_NVP(data) );
  }
};

int main()
{
 cereal::JSONOutputArchive output(std::cout); // stream to cout

 Stuff myStuff;
 myStuff.fillData();

 output( cereal::make_nvp("best data ever", myStuff) );
}
```

which will produce the following JSON:

Explain

```
{
    "best data ever": {
        "data": [
            {
                "key": "imaginary",
                "value": [
                    {
                        "real": 0,
                        "imag": -1
                    },
                    {
                        "real": 0,
                        "imag": -2.9932
                    },
                    {
```

```
                    "real": 0,
                    "imag": -3.5
                }
            ]
        },
        {
            "key": "real",
            "value": [
                {
                    "real": 1,
                    "imag": 0
                },
                {
                    "real": 2.2,
                    "imag": 0
                },
                {
                    "real": 3.3,
                    "imag": 0
                }
            ]
        }
    ]
}
```

If you find yourself attempting to serialize a standard library type and receiving compile time errors about cereal being unable to find an appropriate serialization function, you have likely forgotten to include the type support.

More information on the archives and functions used in the above example can be found in the serialization functions and serialization archives sections of the documentation.

Thread Safety

cereal can be used in a multithreaded environment with some restrictions and precautions, detailed here.

TLDR Version

If you want to be thread safe:

1. Ensure that archives are accessed by only one thread at a time.
2. If you will be accessing separate archives simultaneously, ensure `CEREAL_THREAD_SAFE` is defined and non-zero before any **cereal** headers are included, or modify its default value in `<cereal/macros.hpp>`.

Using Threads Safely

cereal can be used safely with threads with minimal limitations. The most important limitation to be aware of is that individual serialization archives are not designed to be accessed simultaneously from multiple threads. **cereal** expects that access to an archive will happen in a serial manner and may have undefined behavior if an archive is accessed in parallel.

Using multiple archives in parallel

While a single archive should not be used in parallel, it is possible to use multiple distinct archives simultaneously.

cereal uses global objects to track metadata associated with polymorphism and versioning for serialization. To keep **cereal** as light and fast as possible, its default behavior is to assume a single-threaded environment and perform no locking. If you have the need to access distinct archives simultaneously, you will need to define the `CEREAL_THREAD_SAFE` macro to be non-zero:

Explain

```
// Before including any cereal header file
#define CEREAL_THREAD_SAFE 1

// Now include your cereal headers
```

```
#include <cereal/cereal.hpp>
// etc
```

This macro can be found in `<cereal/macros.hpp>` and is detailed in the doxygen documentation. Defining `CEREAL_THREAD_SAFE=1` will cause **cereal** to perform locking on certain global objects used during polymorphism and versioning serialization. This will come with the performance penalties associated with locking a mutex.

Note that you will need to ensure this macro is defined before any **cereal** headers are included in every compilation unit. Alternatively, you can modify the default value for the macro in `<cereal/macros.hpp>` to avoid having to define it manually.

If using **cereal** through CMake, the `THREAD_SAFE` option can be used to add this definition automatically.

@