bakery

Release 0.1

Olivier Hériveaux, Adrien Boussicault, Clément Iatrides, Sofien Bo

Feb 02, 2020

CONTENTS:

CHAPTER

ONE

TUTORIAL

Bakery is a binary data asset loading library. It allows easy creation of data files using a C++-like syntax, provides just-in-time binary compilation and C++ deserialization API.

Bakery has been designed for taking both perks of data description langages, such as XML or JSON, and binary files loading speed. Loading a bakery data file only requires support for binary deserialization in the final program, which is easy to implement and maintain - no need to parse the data files (bakery does it for you).

1.1 Getting started

In the following very quick example, we want a video game program to load settings from a configuration file: screen resolution, fullscreen option, player name and game difficulty. We want the configuration file to be editable with any simple text editor. Also, we don't want to have complex code to parse and load this file in our program code.

Using the Bakery library, the first step is to write a **recipe** file which will specify what fields are to be expected in the settings file:

Listing 1: settings.rec

```
int width;
int height;
bool fullscreen;
string name;
int difficulty;
```

The **recipe** will tell Bakery what fields must be set in the config **data** file. Each field has a defined type. The **data** file can have the following content:

Listing 2: settings.dat

```
recipe "settings.rec";
name = "Gordon";
difficulty = 3;
width = 1024;
height = 768;
fullscreen = true;
```

Once this **data** file has been written, loading is very simple and straightforward:

Listing 3: demo.cpp

```
#include <bakery/bakery.hpp>
...
```

(continues on next page)

(continued from previous page)

```
int width, height;
bool fullscreen;
std::string name;
int difficulty;

bakery::load("settings.dat", width, height, fullscreen, name, difficulty);
```

1.2 What's happening ?

In the example above, the fields defined in the **data** file are not in the same order as specified in the **recipe**. It's not a bug, it's a feature.

In the code loading the data file, the fields are deserialized in the same order as they are declared in the **recipe**, and this is very important! When calling the `load` method, the bakery library builds a binary file *settings.bin* from the **data** file and the **recipe**. The order of the fields written in the binary file is defined by the recipe. The generated binary will look as defined below (for a x64 architecture):

Listing 4: hexdump -Cv settings.bin

```
00000000  00 04 00 00 00 03 00 00  01 06 00 00 00 00 00 00  |................|
00000010  00 47 6f 72 64 6f 6e 03  00 00 00                 |.Gordon....|
```

The first 4 bytes `00 04 00 00` store the width field value (1024 in little endian). The next 4 bytes `00 03 00 00` store the height field value. The next byte `01` is the fullscreen option bool. Then comes the player name: it starts with the value length 6 followed by 6 ASCII characters. Finally, the last 4 bytes `03 00 00 00` are for the difficulty setting.

The builded binary is then open by the program and deserialized into the local variables using the >> operator. Bakery defines deserialization operations for many types, such as `std::string` here.

If the *settings.dat* file is not modified, running the program a second time will not rebuild the binary file and will directly deserialize it. This means loading the data will be really fast since no grammar parsing will happen this time. For this small example the difference won't be noticable, but when using large data files, such as a 3D animated model, this caching mechanism is really efficient.

1.3 Dealing with errors

Loading data can fail if there is an error in the **recipe** or **data** files. When calling the `load` method, Bakery will return a `log_t` object which stores the status of the compilation, and possible error messages. The following code is an example showing how to check and report errors:

```
...
bakery::log_t log =
    bakery::load("settings.dat", width, height, fullscreen, name, difficulty);
if (!log)
{
    std::cout << "Error during settings loading: " << std::endl;
    log.print();
}
```

Alternatively, use `verbose` option to print loading messages in `std::cout`, and `abort_on_error` option to stop program execution when an error is encountered. Thoose option must be set using the `bakery_t` class:

```
...
bakery::bakery_t bak;
bak.set_verbose(true);
bak.set_abort_on_error(true);
// load will call std::abort in case of failure
bakery::load("settings.dat", width, height, fullscreen, name, difficulty);
```

1.4 Improving difficulty field

Currently, the `difficulty` field is defined as an integer, which is not very clear and allows the user setting any arbitrary value. To make the settings file better, we can use **enumerations** to restrict the possible values: here are the changes that can be made in the *recipe* file:

Listing 5: settings.rec

```
enum difficulty_t {
    easy,
    normal,
    hard,
    nightmare
};
...
difficulty_t difficulty;
```

The `difficulty` field can now be defined in the data file this way:

Listing 6: settings.dat

```
difficulty = easy;
```

The `difficulty` field will still be encoded as an `int` in the binary file, so our program should still work as it expects an `int` during the deserialization. Bakery allows deserializing into C++ enumerations as well, but this is not detailed in this tutorial. The `easy` difficulty is encoded as 0, `normal` as 1, `hard` as 2 and `nightmare` as 3. Bakery also allows defining the enumeration values in the *recipe* file like C does, but if not specified default values are set automatically.

When building the settings binary file, bakery will check that the defined value for the `difficulty` matches a member of the `difficulty_t` type. However, for security issues, the value after deserialization MUST ALWAYS be checked against bad input value since an attacker may be able to forge an invalid binary file and bypass compilation. This rule of thumb is valid for any deserialized value!

Bakery has many defined types, supports structures, variants, typedefs, and templates types... This allows creating very rich data formats!

1.5 Saving data

For now we saw how to load a settings data file using Bakery. To go further, we would like to save changes made in the settings during program execution. This involes two operations: serialization and decompilation. The serialization will save the settings in binary file *settings.bin*. Then, Bakery will decompile the binary using the *settings.rec* recipe file and produce a new *settings.dat* file.

```
bakery::log_t log =
    bakery::save("settings.dat", width, height, fullscreen, name, difficulty);
if (!log)
{
    std::cout << "Error while saving settings: " << std::endl;
    log.print();
}
```

CHAPTER

TWO

TYPES

2.1 Standard types

The following table lists all the types natively supported by the Bakery library. It includes standard native types such as `int`, `float`, `bool`, more complex types such as `string`, and also generic types like `list<T>`.

The first column shows the type names to be used in the *recipe* files. The second column shows the equivalent type which can be used for deserialization in the C++ program. Note that one bakery type can have multiple equivalent C++ types: for instance `list` can be deserialized into `std::list<T>` or `std::vector<T>`.

Bakery type	C++ types
`int`	`int`
`short`	`short`
`char`	`char`
`bool`	`bool`
`float`	`float`
`double`	`double`
`string`	`std::string`
`pair<A, B>`	`std::pair<A, B>`
`tuple<T0, T1, ...>`	`std::tuple<T...>`
`list<T>`	`std::list<T>` `std::vector<T>`
`map<K, V>`	`std::map<K, V>`

The binary data size and endianess of the primitive types are architecture dependent, just like C/C++ is. This means you don't have to worry about this when loading data with Bakery, but this also means a compiled binary may not work as intended when copied on another architecture. The recommendation is to always copy the data and recipe files, but not the generated binaries.

Architecture independant types may be added in future versions of Bakery.

2.2 Arrays

Bakery supports fixed and dynamic arrays, as shown in the example *recipe* below:

```
/* Fixed array */
int[3] a;

/* Dynamic array */
int[] b;

/* Multi-dimensional array */
int[2][3] c;
```

Values are assigned as shown in the following *data* file below:

```
recipe "arrays.rec";
a = {1, 2, 3};
b = {1, 2, 3, 4, 5, 6};
c = {{1, 2}, {3, 4}, {5, 6}};
```

Dynamic arrays are equivalent to the `list<T>` type.

2.3 Structures

Structures can be defined in *recipe* files:

```
struct example_t {
    int a;
    float b;
};

example_t data;
```

. . . and set in *data* files:

```
recipe "example.rec";
data = {
    a = 1;
    b = 3.14159265;
};
```

2.3.1 Inheritance

Structure inheritance can be used to add more fields to an existing structure:

```
struct aircraft_t {
    string name;
    int wings;
};

struct jet_t: aircraft_t {
    int reactors;
};
```

(continues on next page)

(continued from previous page)

```
aircraft_t data;
```

```
recipe "planes.rec";
data = {
    name = "A380";
    wings = 2;
    reactors = 4;
};
```

2.3.2 Multiple inheritance

A structure can inherits multiple parents. In that case, the order of declaration is very important because is defines the order of deserialization. In the following structure declaration, the members of `something_t` will be written first in the binary, the members of `aircraft_t` second, and the member `reactors` of `jet_t` third.

```
struct jet_t: something_t, aircraft_t {
    int reactors;
};
```

2.3.3 Templates

Structures supports template type parametization, like C++ does (although the syntax for Bakery is simplified).

```
struct point_t<T> {
    T x;
    T y;
};

point_t<float> point_a;
point_t<int> point_b;
```

Setting the values for template types in the *data* file is transparent:

```
recipe "point.rec";
point_a = { x = 1.5; y = 3.6; };
point_b = { x = 0; y = 10; };
```

Multiple template parameters are also supported by adding more typenames separated with commas. Variadic template parameters are not supported.

2.4 Variants

Bakery supports variant types, which can be deserialized to `std::variant` or `boost::variant`. The following *recipe* shows a variant definition example:

```
variant numeric_t {
    int a;
    float b;
    bool c;
```

(continues on next page)

(continued from previous page)

```
};

numeric_t x;
numeric_t y;
numeric_t z;
```

Assignment in the *data* files is as follows:

```
recipe "variants.rec";
x = a: 5;
y = b: 3.0;
z = c: true;
```

2.5 Default values

Recipes allows defining default values for any variable. When a default value exists, variable assignment in the data file can be omitted and the default value will be written in the compiled binary file.

Note: Although they can serve the same purpose, optional values are different from optional values described in next page.

2.5.1 Example with basic types

Listing 1: settings.rec

```
int width = 1024;
int height = 768;
bool fullscreen = false;
string name = "Player 1";
int difficulty = 1;
```

2.5.2 Example with structures

Listing 2: line.rec

```
struct point_t {
    int x;
    int y;
};

point_t a = {0, 0};
point_b b = {1, 1};
```

2.6 Optional values

A variable can de defined as optional in a recipe. When a variable is optional, assignment in the data file can be omitted.

In the compiled binary, optional values are written using a boolean header indicating if the variable is defined or not. If the variable is defined, it is written in the binary file, otherwise it is missing from the binary and it is up to the serialization to assign a default value when reading the binary. Optional values are therefore more complex to handle in the program, but the produced binaries can be much smaller than when using default values described in the previous page.

A variable cannot be optional and have a default value defined.

2.6.1 Example with basic types

Listing 3: settings.rec

```
optional int width;
optional int height;
optional bool fullscreen;
optional string name;
optional int difficulty;
```

2.6.2 Example with structures

Listing 4: line.rec

```
struct point_t {
    int x;
    int y;
};

optional point_t a;
optional point_b b;
```

CHAPTER

THREE

SERIALIZATION

Bakery comes with deserialization/serialization methods in order to load/save data from/to binary streams. Standard types and classes serialization is already defined by the Bakery library.

For types defined by the library users, serialization operation has to be defined by implementing the template specialization of `bakery::serializer<T>` struct.

3.1 Basic serialization implementation

This section shows how to implement serialization and deserialization for basic structures using a common code for both operations.

Listing 1: demo.rec Recipe

```
struct demo_t {
    int x;
    int y;
    string label;
};
```

```
#include <bakery/serializers.hpp>

struct demo_t {
    int x;
    int y;
    std::string label;
};

template <> struct bakery::serializer<demo_t> {
    template <typename U, typename IO> void operator()(U u, IO & io) {
        io(u.x)(u.y)(u.label);
    }
};
```

This code both implements serialization and deserialization operator, thanks to the template parameters `U` and `IO`. The C++ code defines the `demo_t` structure identically as in the **recipe** file, but this is not mandatory as long as the binary to data mapping is consistent (for example, the names of the members in the **recipe** could be different).

Alternatively, the serialization implementation can be written using some Bakery macros, which hides a little bit the template magics:

```
BAKERY_BASIC_SERIALIZATION_BEGIN(demo_t)
    io(u.x)(u.y)(u.label)
BAKERY_BASIC_SERIALIZATION_END
```

3.2 Advanced serialization implementation

When serialization and deserialization code cannot be the same, the two operators can be defined separately. The following snippet shows how `std::string` serialization operators are defined in Bakery:

```
template <> struct bakery::serializer<std::string> {
    // Deserialization
    template <typename U, typename IO>
        void operator()(std::string & u, IO & io) {
        size_t size = 0;
        io(size);
        u.resize(size)
        for (size_t i = 0; i < size; ++i)
            io(u[i]);
    }

    // Serialization
    // u is const reference
    template typename U, typename IO>
        void operator()(const std::string & u, IO & io) {
        io(u.size());
        for (char c: u)
            io(c);
    }
};
```

CHAPTER

FOUR

C++ API

class bakery_t

Can load or save bakery data files.

Before loading data, options in the bakery can be configured. Directories to be included for recipe files can be added.

After loading data using the bakery, extra compilation information can be retrieved in the state.

Public Functions

bakery_t()

Default constructor.

void **include**(**const** std::string &*dir*)

Includes a directory which may contain recipe files.

Parameters

- `dir`: The directory.

void **include**(**const** std::list<std::string> &*dirs*)

Add a list of include directories.

Parameters

- `dirs`: List of include directories.

const std::list<std::string> &**get_include_directories**() **const**

Return List of directories which may contain recipe files.

void **set_force_rebuild**(bool *value*)

Set or unset force_rebuild switch.

Parameters

- `value`: New value.

bool **get_force_rebuild**() **const**

Return force_rebuild setting.

void **set_verbose**(bool *value*)
Enable or disable verbosity. When enabled, bakery directly prints to stdout information when loading data, and error messages. Verbose option is disabled by default.

Parameters

- `value`: True to enable verbosity, false to disable.

bool **get_verbose**() **const**

Return true if verbosity is enabled, false otherwise.

void **set_abort_on_error**(bool *value*)
Enable or disable abort on error mode. When enabled, any error encountered during data loading will call std::abort to terminate the program in the most possible brutal way. This option is for thoose who don't want to deal with errors themselves.

Parameters

- `value`: True to abort on error, false to continue execution.

bool **get_abort_on_error**() **const**

Return true if bakery aborts on errors, false otherwise.

input_t **load_input**(**const** std::string &*path*, *log_t* &*log*)
Load a bakery data file. Rebuilds the binary cache if necessary, or if the force_build option is enabled. If options for loading data has to be set, use the *bakery_t* class instead.

Return *input_t* for deserialization.

Parameters

- `path`: Path to the datafile.
- `log`: Where error messages are written in case of problem.

template<typename ...**T**>
log_t **load**(**const** std::string &*path*, *T*&... *dest*)
Load a bakery data file and deserialize it in destination variables.

Return Log object containing potential error messages.

Parameters

- `path`: Path to the data file.
- `dest`: Reference to destination variable.

template<typename ...**T**>
log_t **save**(**const** std::string &*dat_path*, **const** std::string &*rec_path*, **const** *T*&... *src*)
Save a bakery data in binary using serialization, and then decompiles it to regenerate a text data file.

Return false in case of error (if abort_on_error is disabled), true if the binary and data files have been written.

Parameters

- `dat_path`: Path to the data file.

- `rec_path`: Path to the recipe file to be used for decompilation.

class input_t

Deserialization class. Non-copyable. Movable.

Public Functions

input_t()

Default constructor. Set stream to null. The input cannot be deserialized after default construction.

input_t(*input_t* &&*other*)

Move constructor.

Parameters

- `other`: Moved instance.

~input_t()

Destructor. Closes the stream.

input_t &**operator=**(*input_t* &&*other*)

Move assignment

Parameters

- `other`: Moved instance.

operator bool() **const**

Return True if Bakery successfully opened the file.

bool **good**() **const**

Return True if Bakery successfully opened the file.

void **set_stream**(std::istream **stream*)

Sets the stream used for deserialization.

Parameters

- `stream`: Stream pointer. This class takes ownership of the pointer.

template<typename **T**>
input_t &**operator>>**(*T* &*t*)

Reads input into t using bakery deserialization.

Return this

Parameters

- `t`: Destination data.

template<typename **T**>
input_t &**operator()**(*T* &*t*)

Reads input into t using bakery deserialization.

Return this

Parameters

- `t`: Destination data.

template<typename **T**>
void **trivial**(*T &dest*)
Trivially loads data by direct stream read.

class log_t
Log filled during the compilation or decompilation process.

Public Functions

log_t()
Constructor

size_t **get_error_count**() **const**

Return Count of error messages.

void **print**() **const**
Print to std::cout all the messages.

std::string **to_string**() **const**

Return A string representing the status. It contains all messages.

void **add_message**(**const** *log_message_t &message*)
Adds a message.

Parameters

- `message`: The message.

void **add_message**(log_message_type_t *type*, **const** std::string *&text*)
Adds a message.

Parameters

- `type`: type of message.
- `text`: Text of the message.

void **error**(**const** std::string *&text*)
Adds an error message.

Parameters

- `text`: Text of the message.

void **warning**(**const** std::string *&text*)
Adds a warning message.

Parameters

- `text`: Text of the message.

const std::list<*log_message_t*> &**get_messages**() **const**

Return List of compilation messages.

void **clear**()
Deletes all the messages from the log.

size_t **size**() **const**

Return Number of messages in the log. To get the number of error messages, use get_error_count.

void **set_rebuilt**(bool *value*)
Set the rebuilt flag value. Called by bakery when loading a data file.

bool **has_rebuilt**() **const**

Return True if the binary has been rebuilt. False if it has been loaded from cache.

operator bool() **const**

Return True if log has no error messages.

bool **good**() **const**

Return True if log has no error messages.

class log_message_t
Holds a message resulting from a compilation (error message, warning message...).

Public Functions

log_message_t()
Default constructor.

log_message_t(log_message_type_t *type_*, **const** std::string &*text_*)
Constructor.

Parameters

- `type_`: Type of the message.
- `text_`: Text of the message.

std::string **to_string**() **const**

Return A string representing the message.

bool **operator==**(**const** *log_message_t* &*other*) **const**

Return true if this has the same text and message type as other.

bool **operator!=**(**const** *log_message_t* &*other*) **const**

Return true if this has a different text or message type as other.

Public Members

log_message_type_t **type**
: Type of the message.

std::string **text**
: Message.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

B