FILE SAVING SYSTEM

VERSION 1.0 – 02/13/2013

1 INTRODUCTION

Software programs always need to save files to remember information after their shutdown. As a file is just a series of zeros and ones, one must be able to know how the file was written in order to read it. Otherwise merely no sense can be given to any digit of the document.

Therefore to read a file, we must know how it was written. If we are developing our own software that reads files, then we know how to read them, since *we* wrote them. But that's not necessarily true. What happens if our software is successful, and we have a new release: will the former release be able to read the files of the new one? And will the new release be able to read the files of the former one?

Over time programs evolve in newer versions, and additional information may be needed to be stored in those files. This will lead to compatibility issues: if one saves additional information in their file, the same file won't be comprehensible by a former version of the software – if nothing has been done to.

This makes your application segmented. Each version will have their own file. This is clearly inconvenient for the end users. Solving this requires a unified way to write your files through every different version of your program.

This document attempts to find a pattern to save portable files. Hence files written by any version of your software will be understandable by any other version of your software (older and newer).

2 TABLE OF CONTENTS

3 DESCRIPTION OF THE PROBLEM

3.1 CURRENT IMPLEMENTATION

Every example is written in C++11, using Qt 5.0. Let's assume we have a Color class that manages colors. We also want to store a color in a file. Hence the class will have a save function and load function.

```
class Color {
                    int m_red, m_green, m_blue;

          public:
```

```
                    // Constructors and other functions...

                    bool save(QString const& file, QString* error = 0) const;

                    bool load(QString const& file, QString* error = 0);

};
```

Basically save and load are implemented as follows:

```
bool Color::save(const QString &file, QString *error) const {
          QFile fOut(file);
          if (fOut.open(QIODevice::WriteOnly)) {
                    QDataStream out(&fOut);
                    // Saves the red channel, the green channel and the blue one.
                    out << (qint32)m_red << (qint32)m_green << (qint32)m_blue;
          }
          else {
                    if (error)
                              *error = fOut.errorString();
                    return false;
          }
          return true;
}

bool Color::load(const QString &file, QString *error) {
          QFile fIn(file);
          if (fIn.open(QIODevice::ReadOnly)) {
                    QDataStream in(&fIn);
                    qint32 red, green, blue;
                    // Loads the channels in the same order.
                    in >> red >> green >> blue;
                    m_red = red;
                    m_green = green;
                    m_blue = blue;
          }
          else {
                    if (error)
                              *error = fIn.errorString();
                    return false;
          }
          return true;
}
```

The first function will save in the file 3 32-bit integers. The first one is the red channel of our color, the next one the green one, and eventually the blue one. Hence our file will contain something like that:

```
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000001110110
```

When the class has to read the file, the class will have to understand those bits. The class manages to do so because it knows that the first 32 bits represent the red channel, the next 32 bits the green one and etc. Hence our class will understand the file as follows:

```
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000001110110
```

Let's now say that we want to store several colors in the file. So we will modify our save and load function so they don't save the color in a single file, but inside another file containing several colors.

```
// QDataStream represents an abstract input/output device: for instance, when we write at the end
// of a QDataStream, I write either at the end of a file, or at the end of a byte array in the
// RAM. Here we save the three channels at the end of an open file.
void Color::save(QDataStream& out) const {
          // Saves the red channel, the green channel and the blue one.
          out << (qint32)m_red << (qint32)m_green << (qint32)m_blue;
}
```

```
void Color::load(QDataStream &in) {
          qint32 red, green, blue;
          // Loads the channels in the same order.
          in >> red >> green >> blue;
          m_red = red;
          m_green = green;
          m_blue = blue;
}
```

That being done, we want to save ten colors in a file. So we have ten instances of our Color class. We create a file and we'll call the save function of every instance.

The file will contain something like this: (30 32-bit integers)

```
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000000011101100
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000000111011000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000001110110000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000011101100000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000111011000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000001110110000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000011101100000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000111011000000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000001110110000000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000011101100000000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000001110110
```

Then again, if we want to load the colors, the software will encounter no problem. The load function knows the first 32-bit integer is red channel, then the green and blue ones. Hence if we call the load function of all our ten instances, they will all manage to read correctly. The first instance will read the first three 32-bit integers, the second the next 32-bit integers and so on:

```
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000000011101100
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000000111011000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000001110110000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000011101100000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000000111011000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000001110110000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000011101100000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000000111011000000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000001110110000000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000011101100000000000
00000000000000000000000010110101000000000000000000000000011010010000000000000000001110110
```

3.2 NEXT-VERSION IMPLEMENTATION

Now our software has evolved. The color class will handle transparent colors as well. Then the class becomes:

```
class Color {
                    int m_red, m_green, m_blue, m_alpha;

          public:
                    // Constructors and other functions...

                    void save(QDataStream& out) const;

                    void load(QDataStream& in);
};
```

The save and load functions are now as follows:

```
void Color::save(QDataStream& out) const {
          // Saves the red channel, the green channel, the blue and the alpha ones.
```

```
        out << (qint32)m_red << (qint32)m_green << (qint32)m_blue << (qint32)m_alpha;
}

void Color::load(QDataStream &in) {
        qint32 red, green, blue, alpha;
        // Loads the channels in the same order.
        in >> red >> green >> blue >> alpha;
        m_red = red;
        m_green = green;
        m_blue = blue;
        m_alpha = alpha;
}
```

We can see that the alpha channel is saved after the three others. We now want to save our 10 instances as we did before. We will get 40 32-bits integers:

```
00000000000000000000000010110101000000000000000000000000000011010010000000000000000000000000011101100
00000000000000000000000100111001000000000000000000000000000101110101000000000000000000000000110100100
00000000000000000000000111011000000000000000000000000000100111001000000000000000000000000010110101000
00000000000000000000001101001000000000000000000000000000111011000000000000000000000000000100110010000
00000000000000000000101110101000000000000000000000000000110100100000000000000000000000000111011000000
00000000000000000010011001000000000000000000000000000101110101000000000000000000000000011010010000000
00000000000000000111011000000000000000000000000000100111001000000000000000000000000010110101000000000
00000000000000001101001000000000000000000000000000111011000000000000000000000000000100110010000000000
00000000000000101110101000000000000000000000000000110100100000000000000000000000000111011000000000000
00000000000001001100100000000000000000000000000010111010100000000000000000000000000110100100000000000
00000000000011101100000000000000000000000000001001110010000000000000000000000000010110101000000000000
00000000000110100100000000000000000000000000001110110000000000000000000000000000100110010000000000000
00000000001011101010000000000000000000000000001101001000000000000000000000000000011101100000000000000
00000000010011001
```

If our current version of the software reads the file, then again there will be absolutely no problem. The software will interpret it as follows (alpha channel is underlined):

```
00000000000000000000000010110101000000000000000000000000000011010010000000000000000000000000011101100
00000000000000000000000100111001000000000000000000000000000101110101000000000000000000000000110100100
00000000000000000000000111011000000000000000000000000000100111001000000000000000000000000010110101000
00000000000000000000001101001000000000000000000000000000111011000000000000000000000000000100110010000
00000000000000000000101110101000000000000000000000000000110100100000000000000000000000000111011000000
00000000000000000010011001000000000000000000000000000101110101000000000000000000000000011010010000000
00000000000000000111011000000000000000000000000000100111001000000000000000000000000010110101000000000
00000000000000001101001000000000000000000000000000111011000000000000000000000000000100110010000000000
00000000000000101110101000000000000000000000000000110100100000000000000000000000000111011000000000000
00000000000001001100100000000000000000000000000010111010100000000000000000000000000110100100000000000
00000000000011101100000000000000000000000000001001110010000000000000000000000000010110101000000000000
00000000000110100100000000000000000000000000001110110000000000000000000000000000100110010000000000000
00000000001011101010000000000000000000000000001101001000000000000000000000000000011101100000000000000
00000000010011001
```

But what happens if the file is read by our former version of the software?

As the former version is not aware that there is an additional channel, it will read the first three integers without skipping the forth. Hence when the second instance will start reading, it will start from the alpha channel, and not the red channel of the second color: (unread digits are in black)

```
00000000000000000000000010110101000000000000000000000000000011010010000000000000000000000000011101100
00000000000000000000000100111001000000000000000000000000000101110101000000000000000000000000110100100
00000000000000000000000111011000000000000000000000000000100111001000000000000000000000000010110101000
00000000000000000000001101001000000000000000000000000000111011000000000000000000000000000100110010000
00000000000000000000101110101000000000000000000000000000110100100000000000000000000000000111011000000
00000000000000000010011001000000000000000000000000000101110101000000000000000000000000011010010000000
```

```
00000000000000000001110110000000000000000000000000100110010000000000000000000000000101101010000000
00000000000000000001101001000000000000000000000000011101100000000000000000000000001001100100000000
00000000000000000010110101000000000000000000000000110100100000000000000000000000001110110000000000
00000000000000000100110010000000000000000000000010110101000000000000000000000000110101000000000000
00000000000000011101100000000000000000000000001001100100000000000000000000000010110101000000000000
00000000000000110100100000000000000000000000011101100000000000000000000000001001100100000000000000
00000000000001011010000000000000000000000001101001000000000000000000000000011101100000000000000000
00000000000010011001
```

We can see that every channel is shifted. The first alpha channel is considered as a red channel, the red channel as a green one and so on.

To summarize: The former version is unable to read the file.

This could be avoided for instance by saving every alpha channel at the very end of the file. But this is absolutely not a good solution:

- It becomes a real headache to decipher the file if every piece of data is scattered all around the file.
- This solution doesn't solve every case. What if we notice that it's useless to store a 32-bit integer for one channel, since every channel is between 0 and 255. So an 8-bit integer would suffice. Hence our files would be 4 times smaller. This solution doesn't cover that case: the former version will read 32-bit integers instead of 8-bit integers.

The purpose of this document is how to save files in such a way that they will be comprehensible by any version of the software (older, and newer).

4 SOLVING THE PROBLEM

4.1 WHAT'S A SOLUTION

Before trying to solve the problem, we should define precisely what is a decent solution to the problem. We want our solution to:

- **Be portable**. We want our files written by our current version of the software to be understandable by any other version (*older and newer*). So the files created by my second version will be readable by my first version and third version of the software. It's usually easy for a document to be understandable by the newer versions, since they know how the file was written. But it becomes tricky when the older versions of the software (which don't know how the future version will write the file) must also be able to read it. We will say that a document is *pseudo-portable* if it can be read by the newer versions, but not the former ones.
- **Be easy to use**. There are dozens of tricks (we will see some later) to write a portable document, or a pseudo-portable document. But they really often tend to make the code very uneasy to write and maintain. This will lead to spaghetti code which is clearly to avoid. Especially because when you need to have portable documents, it means that your software will evolve over time. If the code becomes hard to maintain, it will only slow development time, and make the latest versions of the software much harder to design. Ideally, once the file saving module is programmed, one should *not* have to deal with files, bytes, and even version conflicts anymore.
- **Allow any data model evolution**. During the software development, you may need to store more data than in the former versions. You may also need to save *less* data. Or even to change the format of your

data (e.g. what was stored in 32 bits will now be stored in 8 bits). Or save data in a different order. We want our solution to be flexible so any of those evolutions can be done *easily*, and yet our documents will still be portable.

4.2 SOME SOLUTIONS

In this section we will see some common solutions that try to solve the problem. As a file is just a series of ones and zeros, we need to write the file with certain logic, to be able to read it afterwards (or we won't know how to interpret those digits).

For those solutions to work, the core of the saving system must be the same in every version of the software (we must always read/write the file with the same logic).

4.2.1 ADDING EXTRA DATA AT THE END

4.2.1.1 DESCRIPTION

The first solution we had a glimpse of is to simply add all extra data at the very end of file. For instance, let's consider our color class. We had 10 colors to store in a row. We wrote the 3 channels (red, green, blue) one after another, for the ten colors. Then we added an extra channel. In order to make the file portable, we should write the 10 alpha channels after writing 10 times the three first channels. Or more generally speaking: let E(n) be the extra data the nth version needs to save compared to the n-1th, and D(n) be the whole data that will be saved by the nth version.

So D(n) = D(n-1) + E(n). (The data saved by the n-1th version, plus the extra data we need to save. + is the concatenation operator). And let E(0) be D(0).

So we have D(n) = $\sum_{k=0}^{n} E(k)$.

Then the first version of the software will read D(0) only, which is just what it needs. The software is programmed so it understands D(0).

The next version will read E(0) + E(1) = D(1), regardless of the rest. Once again the version 1 knows how to read D(1), so the file is perfectly understood by the software. And so on. Hence every version will read only what it needs, and there will be no misinterpretations. The file written by any newer version will be comprehensible by the former versions. Besides if a newer version reads an old file, it will see that some data is missing, and it must be able to replace it by default data (for instance, if the alpha channel is missing, we must be able to say that our alpha channel is 255, which means the color is opaque). The document is portable.

4.2.1.2 DRAWBACK

There are several drawbacks in using this solution:

- This solution is not flexible. What happens if tomorrow we don't want to store our colors in 4 32-bit integers, but in a single 32-bit integer? The former versions will always read 4 32-bit integers, which is wrong. So this version does not allow you to modify in any way the format in which you stored data. And this is a serious drawback: as the software is supposed to evolve over time, this format is likely to change

over time. The point of all this was to have a long-term-evolution software, and not being able to even change one data format is a serious disadvantage.

- Besides this doesn't cover the case where you must save *less* data. Let's say that some data of your first version have no use anymore. You don't want to save them (for instance you need to store 9 colors instead of 10). But if you don't save them, the former versions won't be able to load the file. So you have to store dummy data (that will fill the useless data. E.g. you need to store a dummy color) just to counter compatibility issues. Hence this solution doesn't allow you to modify data, nor to remove data.
- This solution is not easy to use. If your software is well programmed, and respecting the object-oriented paradigm, then every data should be located in classes. And there may be classes' instances inside other classes and so on. If ever you add a member in your class, it means that you will have to save this member at the very end of the file! While it will be much more convenient, and much more logic to store it among the other parameters. This will lead to a spaghetti masterpiece. Over time, your code will save pieces of related data all over the file, with no logic, instead of saving them together. It is not a maintainable code.

4.2.2 ADDING VERSION NUMBER INFORMATION

4.2.2.1 DESCRIPTION

We will now see a solution which is easier to use, but is not portable (only pseudo-portable).

This solution adds version information in the file. We can for instance write the current version number at the very beginning of the file (The first 32 bits of the file would be an integer telling the version number).

4.2.2.1.1 HOW TO SAVE THE FILE

The saving procedure would merely write the version number at the very beginning. Then you simply write your data as you would do normally. Just write what is important for your version, regardless of the former versions. This means that you are free to add/remove/modify data compared to the former versions.

4.2.2.1.2 HOW TO LOAD THE FILE

Just read the first 32 bits to have the version number. Then two case scenarios:

1. The version number read is lower or equal to the version number of the running software. Then you can read the rest of the file: the file has been saved by that version or a former one, so you know how to interpret every digit.
2. The version number read is greater than the version number of the running software. It means that the file has been saved by a newer version of the software. So you can't read the file since you don't know how it was written. This means this solution is *not* portable.

4.2.2.2 PROS AND CONS

The main drawback of this solution is that the document is *not* portable, only pseudo-portable (no backward-compatible). But every file can be read by newer versions of the software.

However this solution offers several advantages:

1. The code is much easier to maintain and write than the former solution's. For every new release of your software, you just have to tell how to read the current version files. The code to read the former version files was already in your former software, so you don't have to write it over. You may only have to do some slight changes to adapt the previous load functions to the new data model of the current version. You may, for instance, write a different load function for every existing version of your software. Then, when reading the file, just by reading the version number, you know which load function to call. Hence the code is very clear and easy to maintain.
2. This solution is flexible. In your newer version, you can add, remove, modify, or reorder every piece of data you want. It has absolutely no consequences for former versions (which won't be able to read the file anyway) or the next versions (which will know how you are writing your files right now). This allows you to innovate, doesn't prevent you from changing your data model, with no fear of compatibility issues.

4.2.3 WRITING THE FILE FOR EVERY VERSION

4.2.3.1 DESCRIPTION

The next solution, and the last one before the solution I chose, is portable, allows any data model evolution. But it has a huge downside: it terribly increases the file size.

This solution is an improvement of the previous solution. Instead of writing the current software version, and then the rest of the file, we will write the file contents several times, once for every existing version of the software.

Here is how the file is written:

1. First we write the version number of the first version.
2. Then we write how many bytes the next step will take.
3. Then we write the file contents as we did in the first version of the software.
4. We reiterate with the following version. We stop when we have written the current version.

Then when we have to read the file, we know my version number, so I can directly skip the previous versions bytes, and read the contents of my version. If the file was saved by a former version of the software, then my version number will not be found in the file. So we load the contents of the latest version.

> ***Why would we write the same contents for each version? Shouldn't we just write the file for the first version of the software? There is redundant data.***
>
> Don't forget what our goal is: our software has evolved, and the data model has changed. We may have more data to store, some data we don't need anymore etc. If we only save the file for the first software, we will be stuck with the same data model for ever: we won't be able to save our new data!

4.2.3.2 PROS AND CONS

The main advantage is that our files are portable. However the code becomes harder to maintain. In the previous solution, we had to write a load function for every version. Now we have to write a load function *plus* a save function for every version. When a new release is due, every load/save function may need to be adapted to the new data model. Even though the code will still be organized, it will get longer to develop. Indeed as the version number increases, and the first data models drifting too much from the current one, it will get harder to maintain

(adapt every function). It could even be too much a work to simply think of dropping backward compatibility for the first versions.

The second drawback is of course the file size that increases drastically with the software version.

5 THE CHOSEN SOLUTION – METADATA

5.1 ON THE ROAD TO THE SOLUTION

All the solutions we have seen so far have not been fully satisfying. They were either not portable, either not easy to use, or not flexible.

The main problem we have faced is how to interpret data. From the software point of view, when we open a file, we only see zeros and ones. And we are supposed to understand something! We basically know how to read it, because we know how it was written. For instance we know the first 32 digits represent a version number, then the next 32 digits represent a color, etc…

So we know how to read the file because *the software knows* it. Let's picture it like that: the *manual* to read the file is contained in the software. This means that if tomorrow the file is written in another way, the software won't be aware of the change, and thus won't be able to read the file.

What if the *manual* is now stored in the very file? If we can find a way to write inside the file how to read it, then our document will be portable. The software won't know how to read the file, but will know how to read the *manual* to read the file.

This means that the data model is free to evolve over time. Some pieces of data can be added, removed, modified, reordered, etc… There are absolutely no constraints on the data model. Nonetheless if the data model is free to evolve, the *manual* model must not change over time: i.e. the way you write/read the *manual* must be constant (even though the *manual* itself may change).

5.2 AN EXEMPLE OF MANUAL

Now that we have got the idea of the manual, we still have to find an efficient one. The first idea we could have would be to store at the beginning of the file where to find every piece of data in the file (e.g. its location). This idea is not necessarily good: it may become really headachy to write the manual, and use it efficiently.

I would suggest another kind of manual, and the rest of the document will be focused on that solution. As the idea was introduced, it is natural to think that a manual is an amount of data stored at the beginning of the file. But this is not the only way to do it. Here the manual will be stored throughout the whole file, mixed with actual data. Indeed before writing every piece of data, we will write "a piece of manual". We will here store the name of that piece of data.

What we have called *manual* is usually called *meta-data* (data about data). And it's exactly what we are doing here: we store the name of a piece of data, which is exactly data about data. Manual will from now on be called *meta-data*.

5.3 HOW IT WORKS

The software will read data and meta-data. Meta-data will tell the software how to interpret the related data. But how will the software know how to interpret meta-data? Before, in the software it was written how to interpret data. We will now do the same but with meta-data.

The software will contain a library of meta-data, i.e. a set of every data name it can read. There may be additional data to store in the future (i.e. more meta-data that will be unknown for our software meta-data library), but this is not an issue.

5.3.1 READING THE FILE

Before reading every piece of data, we will read meta-data describing it. That meta-data is supposedly in our software meta-data library. Hence knowing that meta-data we just read, we will know how to interpret the data, i.e. how to read it, and what to do with it.

We may read meta-data that is *not* in our software library. That means that that data has been saved by a newer version of the software. Since we don't know the meta-data, we don't know how to interpret the corresponding data. So we simply *ignore* that data. This makes sense: since it has been saved by a newer version of the software, this data must have a use for the newer versions, but not our current version. So even if we knew how to read it, we wouldn't know what to do with it. If we look at our color example: the first version would load the red, green, and blue channels (with meta-data informing which data is which channel). Then when it comes to read the alpha channel, meta-data would tell this is the alpha channel. But our software doesn't know what's the alpha channel yet. So we simply ignore it. We don't even have a room for alpha channel in our first color class anyway.

Let's say we have now read the whole file, but some data was missing. Here we have two scenarios:

1. The missing data was not essential. We can then give a default value to that data. In our color example, in the second version of the software (which handle alpha-channeled colors), we may read a file written by the first version (containing meta-data, but without alpha channel). The software will load the first three channels without problems. Then it will notice the alpha channel is missing. But we won't throw an error, we will just set the alpha channel to 255, so that our color will be opaque, as it was in the first version of the software.
2. The missing data was essential: we cannot restitute main data without it. Then we have to throw an error, saying that some data is missing. For instance if the red channel is missing in our file, we cannot reestablish the color. We have to throw an error. *Note:* if the file was written cautiously, this case *never* happens, unless it was the developer's will to break compatibility with the former versions.

5.3.2 WRITING THE FILE

Our software contains a large library of meta-data, and some of them may come from older versions of the software. We have to write the whole library - and its associated data - in the file. This means that for every meta-data entry in our library, we will write the meta-data (i.e. the associated data's name) and right after the associated data.

By doing so, we are sure that when a former version reads our file, it will find the data it needs (since its meta-data is in our meta-data library).

This method may save redundant data. This happens when the data format has changed (from an integer to a string for instance), and that we have to store both the string and the integer version, under different name (i.e. labeled with different meta-data). It may sound problematical, but there is no simple way to solve this. If the program expects an integer and receives a string, the file will be misinterpreted and the behavior undefined. Hence it must receive an integer. If we give the thoughts for the newer version that expects a string, then we end up writing down the two versions of the data for compatibility issues.

5.4 WHY THIS SOLUTION SOLVES MOST PROBLEMS

Through this whole paper, we have seen several common issues for file saving. Let's see how they are solved by this solution:

1. **New data has been added to the new version:** Then the newer version of the software will save it, and give it a new label. When former versions read the file, they notice there is new data – because of new data's names that are not in their meta-data library – and they will simply skip those new data. So there are no compatibility issues.
2. **Some data's format has been changed:** This may happen if for instance something that was saved as an integer is now saved as a string, or as 32-bit integer and new a 64-bit integer. The software will have to save the versions of that data, in every different format. And each format must have a unique label/data name. The former versions will be able to read the older format since they will recognize their data name. And like above they will skip the new ones since they won't recognize their data name.
3. **Some data has been removed:** Then the newer version of the software may simply don't save it. The former versions will notice that some data has been removed. Then two things will happen: the software can run without knowing that data. Then it will give the missing data a default value and keep going. Or that data was mandatory. Then the software throws an error saying the file is miswritten. In this case there is a compatibility issue, but it was of the developer's will to break compatibility when he decided not to store that mandatory data. *Note:* This case is extremely unlikely to happen. Because it means that mandatory data of a first version has become useless in the newer versions of the software. This means that the software has drastically changed since the first version.
4. **Some data is not saved in the same order:** This was an issue before. As we did not have a pattern to interpret data, we interpreted them according to their position in the file. The first data would be interpreted as something, the next one to another thing and so on. And the order in which data was written had to be respected, or we would misinterpret everything.
   This is no longer the case with that solution, thanks to meta-data. The order doesn't matter anymore, we don't need it to interpret data.
   It may happen that the way to interpret some data is related to another data we read before. This is not an issue with this solution. Indeed, we first read every datum and then we interpret it: we read the whole file, and end up with every piece of data labeled by their name. Then if one piece of data must be interpreted before the rest, we just have to *interpret* it first. Hence, where this data was actually located in the file doesn't matter.

Hence that solution is portable and allows any data model evolution. We will now see an implementation that will make it easy to use.

6 IMPLEMENTATION

In this last part we will see an implementation of that method. We assume here that we fully use the oriented object paradigm. The data we want to read/write is a class instance. We want to save and load several or all attributes of that class. We will call that class *Entity*.

6.1 SEPERATING TASKS

The *Entity* instance must not know how to save or load itself in/from a file. Indeed, a class is an object, which achieves a specific purpose. And unless that purpose is to deal with files, the class must not know, by its nature, what is a file. The class does what it was designed for and nothing else, and this is how it should always be. We will not break this rule in that solution.

This means that the *Entity* class will not have a save or load function. We want to separate tasks for a better code organization. Hence we will create a new class whose purpose is to save a class instance in a file, or to load one from a file. We will call that class *Manager*.

6.2 DATA FORMAT TO SAVE/LOAD

Let's now have a look at what will be written in the file. For each data we record, we must save its name (meta-data). Doesn't that remember you a specific data structure? Hash tables!

Hash table

Hash tables are arrays that associate an index to a value, hence they are called associative arrays. They are basically arrays but their indexes are not integers, but may be of any type. Usually indexes are strings. They are useful to associate a string to a particular value.

Example:

```
std::unordered_map<std::string, int> hash;
// hash table whose indexes are strings, and values are integers;
hash["0th"] = 0;
hash["1st"] = 1;
hash["2nd"] = 2;
hash["3rd"] = 3;
hash["4th"] = 4;
hash["5th"] = 5;
int fifth = hash["5th"]; // 5
```

This means that what we will store in the file is basically just a hash table. Indexes will be strings (our data names), and the values will be "data". Since those data may be of any type, let's say they are arrays of bytes (since everything in our computers is basically just an array of bytes).

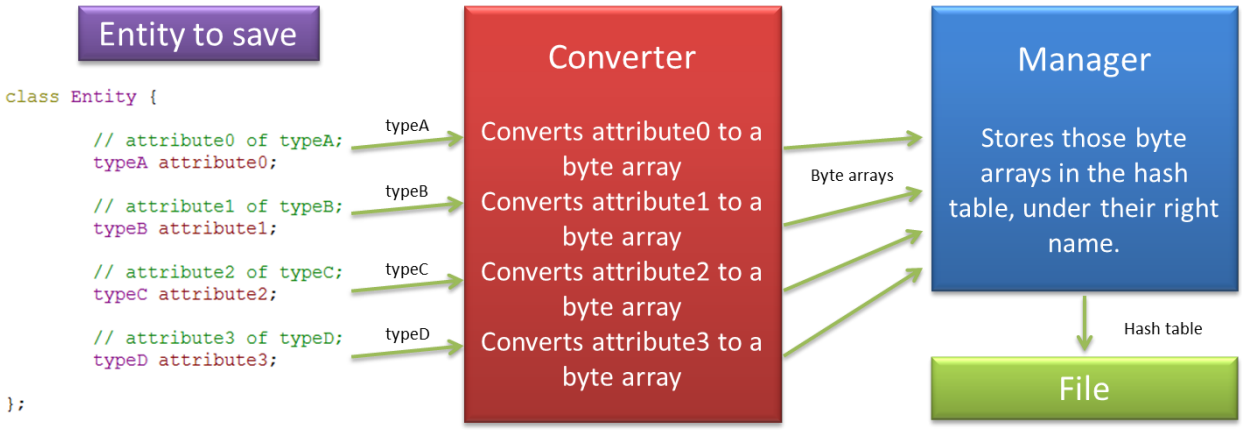
So we want to save a *hash table* whose indexes are *strings*, and values are *byte arrays*.

Now that we know the structure of what we want to save, we need to figure out how to get that from the class attributes we want to save to the hash table. Basically the values of our hash tables are the exact attributes of our class, and the indexes are their name. Those names are arbitrary, and we will give a name to each attribute later. What's important is that each attribute must have a different name.

6.3 CONVERTER

We will now see how to convert our class attributes to byte arrays, so we can store our attributes in the hash table and save them in a file (or the other way around, load the hash and map the arrays to the attributes). We *do* need to make the conversion from an attribute to a byte array. Because even if in our computer's RAM, attributes are just bytes, copying those bytes in the hash table may lead to serious issues. For instance in C or C++, attributes may be pointers to something, and we surely don't want to save the address of what's pointed, but rather its content.

It's not the role of our class to convert those attributes to byte arrays, nor the other class (*Manager*) that stores the hash table in a file. So we need a third class that will do the job: *Converter*. We come up with that organization:



Every attribute is converted to a byte array by *Converter*, and then it's stored in the hash table of *Manager*. Finally *Manager* saves that hash table in the file. Of course it's the other way around when it comes to load the file.

The next question we must ask is how does *Manager* know the name of every byte array?

6.4 MANAGER

The answer is simple: we will tell *Manager* the name of every data we must save. This means that *Manager* will know our entire meta-data library.

Along with telling it what the data names are, we will tell it where to find those data. In order to convert every attribute to byte arrays, *Converter* has a member function for every attribute of the *Entity* class that will do the conversion. Consequently when we tell *Manager* what are the meta-data, we will tell it which *Converter* member function to call to get the associated data.

Let's see an example. Our *Entity* class will be the Color class (with the red, green and blue channels).

```
class Color {
                    int m_red, m_green, m_blue;

          public:
                    // Constructors and other functions...
};
```

Our *Converter* class:

```
// We need one converter class for every Entity class there is to save
class ConverterForColor {

          // Converter has an access to the instance it has to save/load.
          Color* m_entity;

          public:
                    // Constructors and other functions...

                    // reads the red channel attribute of Color, and write it in a byte array
                    bool readRedChannel(QByteArray& red);
                    // same with the other channels
                    bool readGreenChannel(QByteArray& red);

                    bool readBlueChannel(QByteArray& red);

                    // writes the red channel attribute of Color from a byte array
                    bool writeRedChannel(QByteArray const& red);
                    // same with the other channels
                    bool writeGreenChannel(QByteArray const& red);

                    bool writeBlueChannel(QByteArray const& red);

};
```

Now we can see more precisely what to tell *Manager*. We need to specify the data names, and how to access the associated data. Therefore, we will give it that list:

1. There a piece of data called "*color.red*", you can read it through the `readRedChannel` function of the converter, and you can write it through `writeReadChannel`.
2. Another one called "*color.green*", readable through `readGreenChannel`, writable through `writeGreenChannel`.
3. Last one called "*color.blue*", readable through `readBlueChannel`, writable through `writeBlueChannel`.

The names given are completely arbitrary. Data names don't matter that much, as long as they are all different.

Then, knowing that, when it comes to save a file, *Manager* will know what needs to be saved. It will call every function we previously entered, and save the byte arrays it got under the proper index in its hash table. It will eventually save the file.

Loading a file is a little bit more complicated. *Manager* will go through the following steps: (this algorithm is not complete, and does not cover every case. A full version is available at the end)

1. *Manager* will load the hash table.
2. It will then go through the whole hash table indexes, and if it recognizes a data name (such as "color.red"), it will call the associated function of the Converter.
    a. Converter will convert the byte array back to its original type.
    b. And will set that read value to the *Entity* instance.

Actually the load algorithm is more complex than that. What happens if my data format has changed? We've seen before that we must save every format in the file, with a different name. But when it comes to load the file with the current version, I recognize every format since they are in my meta-library library. So I must choose one from the others (or I will load several times the same data, according the simple algorithm written above).

We will fill that gap in the next part.

6.5 IMPLEMENTATION DETAILS

Before looking at how to implement *Manager* efficiently, let's go back to *Converter*.

6.5.1 CONVERTER

We have seen that for every *Entity* class we have to save, there must be a *Converter* class. And this looks like a real pain: having to write a duplicate of the class that will only convert data from a type to another. However, depending on the language you implement the system in, there may be several tricks to avoid doing that.

Let's take the C++ case, since everything has been written in C++ so far. In a perfect world, what you would like to do is tell *Manager* which attributes to save, and under which name. Then *Manager* will figure out what function to call, or even generate the functions on its own. This is possible thanks to template meta-programming, or even lambda functions of C++11. With those tools, you can tell C++ to *generate* the Converter class, with every converting function *automatically*. Since those functions are generated automatically, they won't offer an advanced conversion process. But if it happens that an attribute needs a specific treatment (e.g. a pointer) then you can still write the converting function for that attribute only, and C++ will generate the rest.

Anyway, you can think on your own implementation. Depending on your language, it may be easier to simply skip that class, and merge it with *Manager* or even *Entity*.

6.5.2 MANAGER

We have finally reached the last step, how to implement *Manager* efficiently! We have had a first glimpse before, but that implementation wasn't satisfying. All we did was supplying a list of data names, and how to access those data. We need to provide more data about those meta-data.

Do you remember that there were three possible cases about data evolution? Either we add more data from a version to another, either we remove data, either we change the data format.

That simple implementation worked very well if we save more data, but doesn't do the job correctly in the two other cases.

6.5.2.1 DATA REMOVAL

For instance, if one piece of data is missing when we load the file, how can we tell if that data was *mandatory* or if we can give it a *default value*?

Here we are, we already have two more things for Manager. For each piece of data, it must know if that data is mandatory. And if it is not, it must know a default value for that data (that it will send to *Converter* like any other data). The mandatory information is a boolean, and the default value a byte array (since it will be sent to *Converter*).

6.5.2.2 DATA FORMAT CHANGEMENT

When the format has changed, we need to save every different format in the file. So when it comes to load the file, how to choose one format of the same data if we recognize more than one?

We must establish a hierarchy among the same data whose format has changed. If we give that hierarchy to *Manager*, he will then be able to choose one.

However creating a hierarchy is not as easy as it may seem. Some data may have their format changed, but some data may be merged or split into several others. For instance in the first case, I have a 32-bit integer that stores the red channel, and later I want to use an 8-bit integer. And in the second case, I stored the three channels of my color altogether, under the same name, and I know want to save every channel separately. Hence, with the allowance of data merging, the problem just got even more complicated.

Let's think it through: we have several pieces of the same data whose format has changed. When that data is split, then they do not represent the original data individually. For example if the color is split in 3 pieces of data, one for each channel, then each channel individually doesn't represent a color.

We will continue on the Color example for a while. What we want is an order of preference between every possible way to store a color. In our example we can either store the three channels altogether (data name: "*color*"), or individually (data names: "*color.red*", "*color.green*", "*color.blue*"). We prefer when they are individually saved. So what we must tell *Manager*, when it comes to read the file, is:

1. If you find the data names "*color.red*", "*color.green*", AND "*color.blue*", then load them.
2. Otherwise load "*color*".
3. (If "*color*" is not found, then check if the color is mandatory. If not, give the color a default one.)

Here we have our hierarchy: we must send to *Manager* a list of groups of data names, sorted by order of preference. Each entry is a group of data names (here "*color.red*", "*color.green*", "*color.blue*" for the first entry, and "*color*" for the second one), that, merged together, represent the same information. Then *Manager* will go through the list. For each entry it will check if every name of the group is in the file. If they are, then it loads them, otherwise it goes to the next entry.

But we also forgot that some pieces of data are mandatory, or have default values. We may even extend the concept: we may say whether such a list of data is mandatory, or if it has a default value. If we combine both, here is what we come up with to read those data from the file:

1. Open the list of groups of data names.
2. For each group of data:
    a. If every mandatory data is in the file, AND there is at least one data in the file from that group (*if we don't write that second condition, and nothing is mandatory in that group, then it will give a default value to everything without looking at the next entry*) THEN load that data, and give default values where data was missing. END LOOP.
    b. Otherwise go to the next group of data.
3. If nothing has been found at all, check if that list is mandatory. If it is, throw an error, otherwise give a default value to that data.

6.5.2.3 LOADING ORDER

We have solved every possible problem of data model evolution. There is now just a tiny little thing to do before the end. We had said before that when it comes to load a file, some data may need to be interpreted (i.e. sent to *Converter*) before others.

Then we must tell *Manager* which data to interpret first, or an order in which it will interpret data. Data can be a single piece of data (one data name), or a list of groups of data names (data whose format has changed).

Now *Manager* is ready to be used!

Just for convenience, I would advise to create a *Manager_Entity* class for every *Entity* class we need to save. That *Manager_Entity* will inherit from *Manager*, and will in its constructor tell *Manager* the whole meta-data library. Hence you only have to write the meta-data library once for every *Entity* class.

We have seen every aspect of the implementation. We will now see the algorithms.

6.6 SAVE ALGORITHM

Here is the save algorithm. As we want it to be reusable, it is not written in C++, but in pseudo-code. It basically saves everything.

```
// out and error will be modified by the function call.
bool save(ByteArray& out, String& error) {
     let rawdata be of type Hash(index: String, value: ByteArray);
     for each entry meta of the meta-data library {
          if meta is a single piece of data {
               let raw be of type ByteArray;
               read raw from Converter; // the member functions to call are stored in meta;
               if Converter threw an error {
                    write the error in error;
                    return false;
               }
               rawdata[meta.name] := raw;
          }
          // meta contains a list of groups of data names.
          else for each entry group of meta {
               for each entry single_meta of group {
                    let raw be of type ByteArray;
                    read raw from Converter; // member functions to call are stored in single_meta
                    if Converter threw an error {
                         write the error in error;
                         return false;
                    }
                    rawdata[single_meta.name] := raw;
               }
          }
     }
     write rawdata at the end of out;
     return true;
}
```

6.7 LOAD ALGORITHM

```
// error will be modified by the function call.
bool load(ByteArray out, String& error) {
     let rawdata be of type Hash(index: String, value: ByteArray);
     read a hash table from out, save it in rawdata;
     for each entry meta of the meta-data library {
          if meta is a single piece of data {
               if rawdata contains (meta.name) {
                    send raw to Converter; // the member functions to call are stored in meta;
                    if Converter threw an error {
                         write the error in error;
                         return false;
                    }
               }
               else if meta is mandatory {
```

```
                    error := "File error: some mandatory fields are missing. ("+ meta.name + ").";
                    return false;
               }
               else if meta has a default value {
                    send the default value to Converter.
                    if Converter threw an error {
                         write the error in error;
                         return false;
                    }
               }
          }
          else {
               let hasSaved be false;
               for each entry group of meta {
                    let containsAllMandatory be true;
                    let containsAny be false;
                    for each entry single_meta of group {
                         if rawdata doesn't contain (single_meta.name) AND single_meta is mandatory
                              containsAllMandatory = false;
                         if rawdata contains (single meta.name)
                              containsAny := true;
                    }
                    if (containsAllMandatory AND containsAny) {
                         hasSaved := true;
                         for each entry single meta of group {
                              if single_meta is mandatory {
                                   send rawdata[single meta.name] to Converter;
                                   if Converter threw an error {
                                        write the error in error;
                                        return false;
                                   }
                              }
                              else if single meta has a default value {
                                   send the default value to Converter;
                                   if Converter threw an error {
                                        write the error in error;
                                        return false;
                                   }
                              }
                         }
                         end loop;
                    }
               }
               if not hasSaved {
                    if meta is mandatory {
                         error := "File error: some mandatory fields are missing. ("+ meta.name
                         + ").";
                         return false;
                    }
                    else if meta has a default value {
                         send the default value to Converter.
                         if Converter threw an error {
                              write the error in error;
                              return false;
                         }
                    }
               }
          }
     }
     return true;
}
```

7 CONCLUSION

We have seen a way to handle file versioning efficiently. Saving extra data (meta-data) in a file allows us to read it by any version of software. But this solution only showed a glimpse of what meta-data are capable of. We could store much more meta-data in our files and have much smarter reading/writing systems.

In all our examples, we have written and read only primitive types (e.g. integers, strings). But what if we have custom types to save? Well since this solution saves one class in a file, if your class has an attribute with a complex type, just use this solution recursively. It is easy to implement: in *Converter*, when you have to convert your complex attribute to a byte array, just call your attribute's *Manager* and it will save your attribute instance in a byte array (assuming that *Manager* can save/load from a file and a byte array). It is exactly the same thing to load your attribute. *Converter* has to call your attribute's Manager and load the attribute from the byte array *Converter* received. This solution, allied with recursion, is extremely powerful.-

This solution drastically reduces development time and file handling issues. When a class is to be saved in a file, we just have to tell *Manager* what are the attributes to save. That's it! We do not handle files anymore, nor bytes and other convoluted things, *Manager* does it for us. If we have an efficient *Manager* API, all we do is tell *Manager* to save a class instance, or to load it.

The only drawback of that solution is that we save more things than what we actually need. But so is the price to pay to have a portable file system, since it is impossible to save only what's necessary and still have a full compatibility.