Object Oriented Software Design II

Real Application Design

Christian Nastasi
http://retis.sssup.it/~lipari
http://retis.sssup.it/~chris/cpp

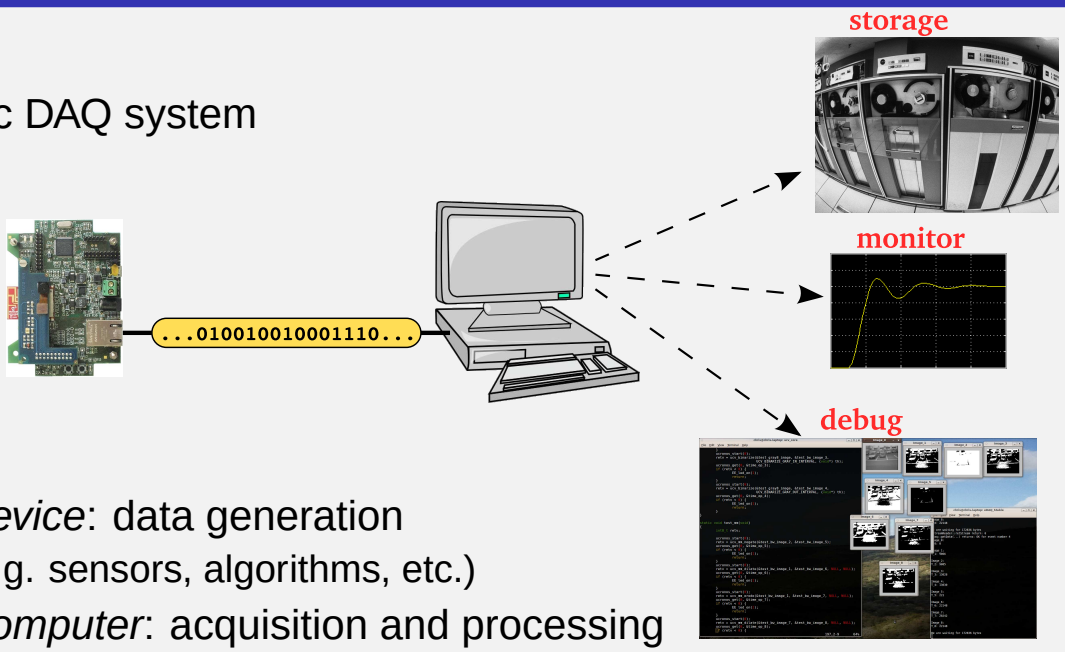
Scuola Superiore Sant'Anna – Pisa

March 27, 2012

Outline

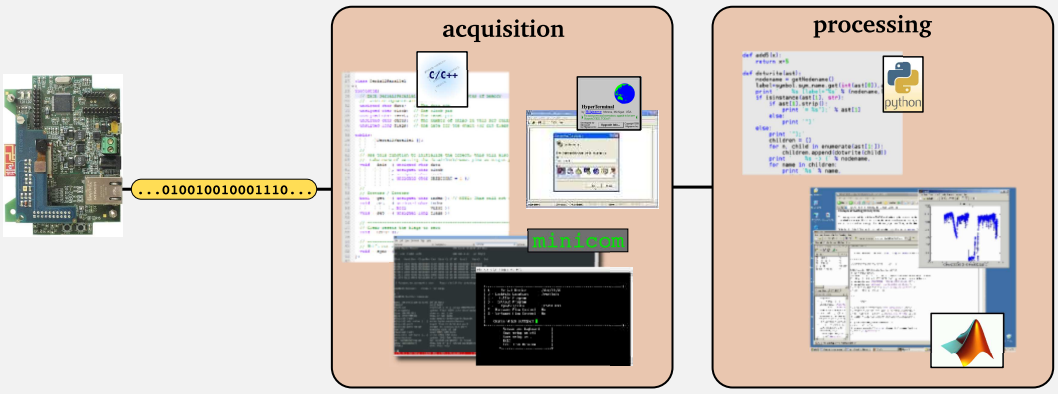
Data AcQuisition System

A basic DAQ system



- *Device*: data generation
  (e.g. sensors, algorithms, etc.)
- *Computer*: acquisition and processing
  - storage
  - visualization
  - post-processing, sometimes on-line processing

Solution 1: custom application

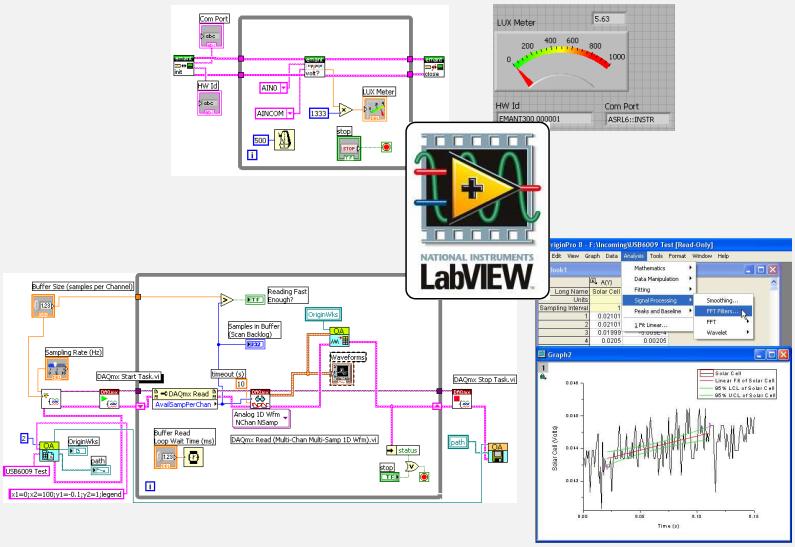


- Simple ... (apparently)
- Useful for basic log (e.g. *printf*s)
- Stream specific
- Data specific
- Reconfiguration requires rewriting!

unmanageable: eventually too many custom applications

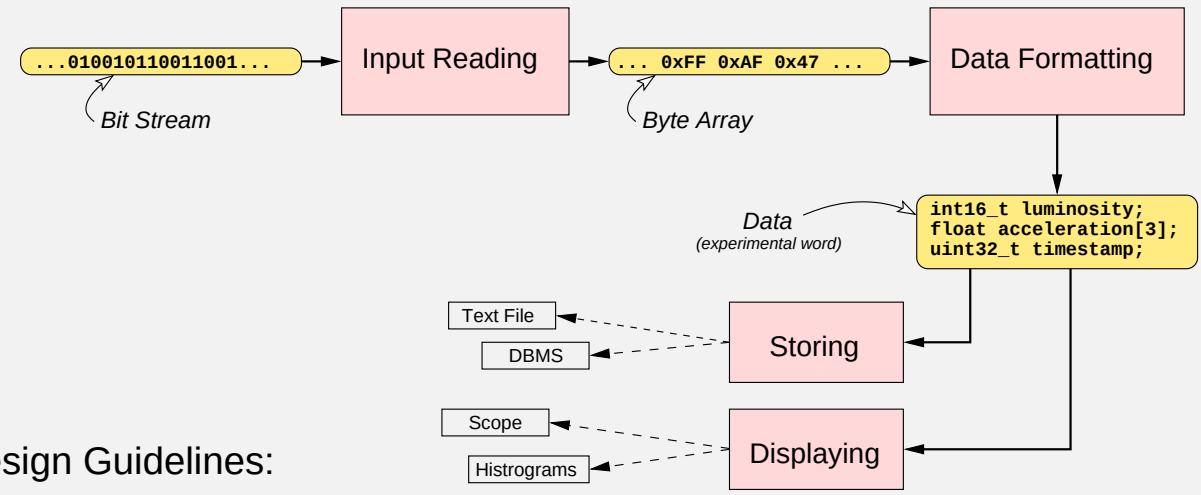
Solution 2: generic application

LabVIEW



- GUI and Graphic Language
- Extremely intuitive
- Flexible
- Highly integrated with several HW platforms
- Used for very complex DAQ: e.g. particle physics experiments
- Commercial SW

Our goal: simple but generic solution



Design Guidelines:

- Generic Data Types (e.g. integers, floats, structured data)
- Generic Data Stream (e.g. serial port, ethernet, etc.)
- Multiple storages and monitors
- **Extensibility**
- **Run-Time (Re)Configuration**

Our application: duck-lab

duck-lab: an open-source DAQ framework

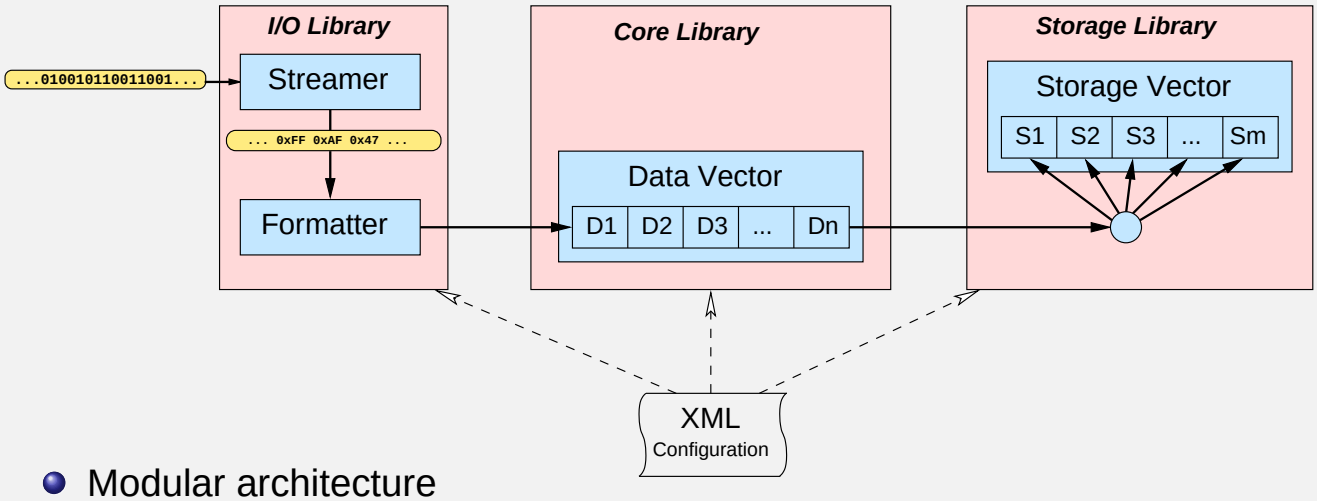
Basic features:

- acquisition from different streams (serial port, socket, parallel port, etc.)
- different storages (text-files, ROOT data framework, etc.)
- run-time configuration through XML file
- **currently under development!**
- **projects available for this course!**

Availability (LGPL license):

- Project (temporary) home page: `http://retis.sssup.it/~chris/duck-lab`
- Mercurial revision control: `https://rtn.sssup.it/hg/duck-lab` (`hg clone https://rtn.sssup.it/hg/duck-lab`)

duck-lab architecture

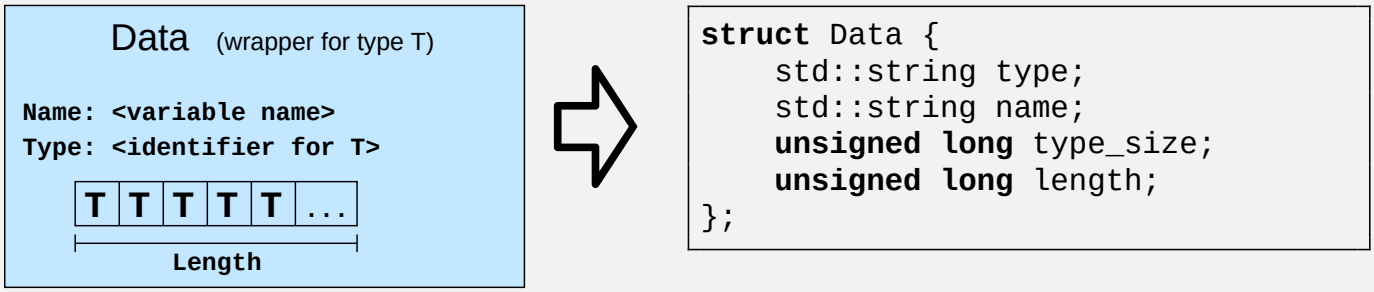


- Modular architecture
  - Core module: Data types and helper classes/functions
  - I/O module: acquisition and formatting
  - Storage module: storing information in several manners
  - Display module: not implemented yet
- Configuration Language: XML file

Data hierarchy

Any *Data* is characterized by:

- *type*: the type of the Data (e.g. integer, float, structured)
- *type-size*: the size in byte of a type (bytes required to create a type-element)
- *name*: the name of the Data instance (much like a C-variable name)
- *length*: how many type-elements are associated to Data (much like length of a C-array)




```
struct Data {
    std::string type;
    std::string name;
    unsigned long type_size;
    unsigned long length;
};
```


Data hierarchy

Being a little more C++ ...

```
class Data {
public:
    Data(const std::string& t, const std::string& n, unsigned long s, unsigned long l) :
        type(t), name(n), type_size(s), length(l) {};
    inline std::string get_type(void) const { return type; };
    inline std::string get_name(void) const { return name; };
    inline unsigned long get_type_size(void) const { return type_size; };
    inline unsigned long get_len(void) const { return length; };
    inline unsigned long get_tot_size(void) const { return length * type_size; };
private:
    std::string type;
    std::string name;
    unsigned long type_size;
    unsigned long length;
};
```

- Where do we store the actual data?
- We derive `Data` into concrete data classes to do that!

Concrete `Data` example

Example: 8-bit signed integer (aka `char`)

```
class DataInt8 : public Data {
public:
    DataInt8(const std::string& name, unsigned long len) :
        Data("int8", name, 1, len)
    {
        actual_data = new char[len];
    };
    ~DataInt8()
    {
        delete[] actual_data;
    };

    char* actual_data;
};
```

Initializes the base class: `name` and `len` are forwarded while `type` and `type-size` are explicitly set.

Allocation of resources

Deallocation of resources

Specific implementation (array of `char`)

Concrete `Data` example

A more complex example:

- user-defined data (structured data)
- an RGB-image class

```
class DataImageRGB : public Data {
public:
     DataImageRGB(const std::string& name, unsigned long len, unsigned width, unsigned height) :
         Data("ImageRGB", name, width * height * 3, len)
    {
        // Allocate internal representation
        // for RGB image with resolution 'width' x 'height'
        // ...
    };
    ~DataImageRGB()
    {
        // Deallocations...
    };

    // ...
};
```

Constructor can be specific for each derived class

Question

- Any mistake so far?
- Let's consider the following example.
- Where is the problem?

```
Data* data_creator(/*could take params...*/)
{
    // might decide what Data has to be created
    //...
    return new DataInt8("test_data3", 5);
}

int main(void)
{
    {
    DataInt8 d1("test_data1", 1);
    DataInt8 d2("test_data2", 10);
    }
    Data* d3 = data_creator();
    delete d3;

    return 0;
}
```

Function that creates `Data` object according to some configuration

Problem: The memory allocated here is not properly released!

Object `d1` and `d2` are automatically deleted when out of scope

Object `d3` has been allocated on heap (with new), has to be manally deleted

Solution: The destructor of the `Data` must be `virtual`!

`./examples/07.real-application_inheritance-examples/DataExample1.cc`

Don't forget virtual destructor

- Define a virtual (empty, if necessary) destructor in the base class
- All classes derived from `Data` shall automatically have a virtual destructor

```
class Data {
public:
    //...
    virtual ~Data() {};
};
```

```
Data* data_creator(/*could take params...*/)
{
    // might decide what Data has to be created
    //...
    return new DataInt8("test_data3", 5);
}

int main(void)
{
    {
    DataInt8 d1("test_data1", 1);
    DataInt8 d2("test_data2", 10);
    }
    Data* d3 = data_creator();
    delete d3;

    return 0;
}
```

This performs a polymorphic function call. The appropriate destructor `~DataInt8()` shall be called instead of the base class one.

Generalization of Data

- Some classes might not be “interested” in all the members of `Data` .
- We could add one more layer of abstraction to the `Data` hierarchy.
- We defined `Metadata` as generalization of `Data` :

```
class Metadata {
    Metadata(const std::string& t, unsigned long s, unsigned long l) :
        type(t), type_size(s), length(l) {};
    virtual ~Metadata() {};
    inline std::string get_type(void) const { return type; };
    inline unsigned long get_type_size(void) const { return type_size; };
    inline unsigned long get_len(void) const { return length; };
    inline unsigned long get_tot_size(void) const { return length * type_size; };
protected:
    std::string type;
    unsigned long type_size;
    unsigned long length;
};
```

```
class Data {
public:
    Data(const std::string& t, const std::string& n, unsigned long s, unsigned long l) :
        Metadata(t, s, l), name(n) {};
    inline std::string get_name(void) const { return name; };
protected:
    std::string name;
};
```


More basic data types

- We have done Data for `int8`.
- We want to support other basic types ...
- but we don’t want to rewrite code. Suggestions?

Template

```
template <class T>
class DataTemplate : public Data {
public:
    DataTemplate(const std::string& type, const std::string& name, unsigned long len) :
        Data(type, name, sizeof(T), len)
    {
        actual_data = new T[len];
    };
    ~DataTemplate()
    {
        delete[] actual_data;
    };

    T* actual_data;
};
```

More basic data types

```
#include <stdint.h>

class DataInt8 : public DataTemplate<int8_t> {
public:
    DataInt8(const std::string& name, unsigned long len) :
        DataTemplate<int8_t>("int8", name, len) {};
};

class DataUInt16 : public DataTemplate<uint16_t> {
public:
    DataUInt16(const std::string& name, unsigned long len) :
        DataTemplate<uint16_t>("uint16", name, len) {};
};

//...

class DataFloat32 : public DataTemplate<float> {
public:
    DataFloat32(const std::string& name, unsigned long len) :
        DataTemplate<float>("float32", name, len) {};
};

// ...
```

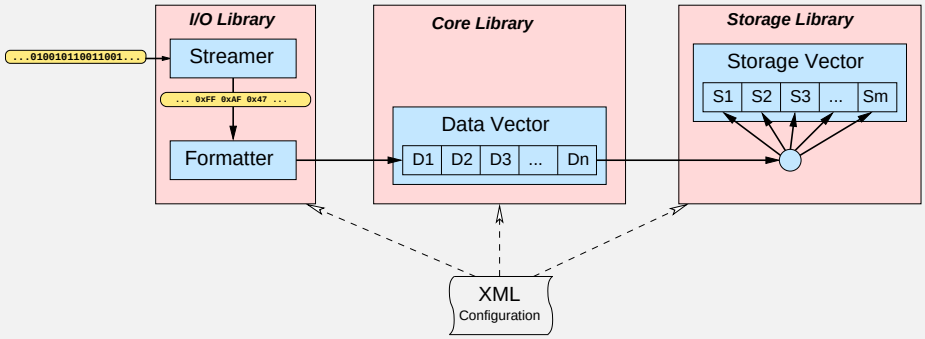

What's next?

- We have a first "generic-data" support
  - We have defined a hierarchy for `Data`
  - We have concrete data classes for basic types (using templates)
  - We can create user-defined concrete data classes
- To improve the definition of `Data` we have to consider
  - how and by whom it is created
  - how and by whom it is manipulated (used)
  - how and by whom it is released
- Study the `Data` lifetime

Data Vector lifetime

How the Data Vector is used in the application:

- someone creates the (concrete) Data objects
- someone formats each (concrete) Data objects
- someone store the (concrete) Data objects



The *Formatter* seems to be the starting point:

- creates and holds the data vector
- receives the raw bytes from the *Streamer*
- formats the data vector from the received bytes

The `FormatHandler` class

- The *Formatter* module is implemented by the `FormatHandler` class
- The existence of the Data Vector is bound to the `FormatHandler` class

```
class FormatHandler {
public:
    FormatHandler();
    ~FormatHandler();
    void format(const uint8_t* rawdata,
                unsigned long len);
private:
    std::vector<Data*> data_vector;
};
```

The Data Vector is created when this object is constructed

The Data Vector is released when this object is destructed

This method formats the Data Vector according to the input raw data (bytes)

The container for the Data Vector

Data Vector creation

The XML specifies what is the Data Vector

```
<message_description>
   <data data_type="uint16"  data_name="data_name_1" data_len="2"/>
   <data data_type="float32" data_name="data_name_2" data_len="1"/>
   <data data_type="int8"    data_name="data_name_3" data_len="4"/>
</message_description>
```

Let's assume (for now) to have a way to parse the XML in a simple C++ container:

```
struct XmlDataElement {
    std::string type;
    std::string name;
    unsigned len;
};

std::vector<XmlDataElement> parse_xml_data(/*might have params..*/)
{
// creates the output vector
}
```


Data Vector creation/destruction

```
using namespace std;

FormatHandler::FormatHandler()
{
    vector<XmlDataElement> config = parse_xml_data();

    vector<XmlDataElement>::const_iterator it;
    for (it = config.begin(); it != config.end(); ++it) {
        Data *d;
        if (it->type == "int8")
             d = new DataInt8(it->name, it->len);
        else if (it->type == "uint16")
             d = new DataUInt16(it->name, it->len);
        else if (it->type == "float32")
             d = new DataFloat32(it->name, it->len);
        data_vector.push_back(d);
    }
}

FormatHandler::~FormatHandler()
{
    vector<Data*>::iterator it;
    for (it = data_vector.begin(); it != data_vector.end(); ++it)
        delete *it;
}
```

Creates the Data Vector. Notice that the default constructor for `data_vector` is called at this point, creating an empty container.

This pointer **is NOT initialized** if `type` is wrong!

We append a `Data*` to the Data Vector.

Destroys the Data Vector.

The `data_vector` destructor is called here, but this does not destroy the data-objects because it contains pointers.

- But something is missing...

Data Vector creation/destruction

Let's consider the following example (see `./examples/07.real-application_inheritance-examples/DataExample1.cc`)

```
int main(void)
{
    {
    FormatHandler formatter;
    } // ~Formatter() is called here!

    return 0;
}
```

- When `formatter`'s scope terminates the destructor is called
- The `data_vector` will contain an invalid (non-initialized) `Data*`
- Then `delete` is called on invalid pointer!

Good advice

When designing your application always check conditions on input values.

- Either we know that `parse_xml_data()` returns only valid types,
- or we check before appending to the Data Vector.

```
for (it = config.begin(); it != config.end(); ++it) {
    Data *d;
    if (it->type == "int8")
         d = new DataInt8(it->name, it->len);
    else if (it->type == "uint16")
         d = new DataUInt16(it->name, it->len);
    else if (it->type == "float32")
         d = new DataFloat32(it->name, it->len);
    else {
        cout << "Unknown data type = '" << it->type << "' ! Skipping" << endl;
        continue;
    }
    data_vector.push_back(d);
}
```

Other possible inconsistencies (depending on application requirements):

- data length set to zero
- data name not given (empty)
- data name redundant

The `FormatHandler` class

- We have done creation/destruction
- Let's do `Data` formatting

```
class FormatHandler {
public:
    FormatHandler();
    ~FormatHandler();
    void format(const uint8_t* rawdata,
                unsigned long len);
private:
    std::vector<Data*> data_vector;
};
```

Formatting method

The `format()` is called by the *Streamer* when new data are available

Data Vector formatting

A first implementation of the formatting method

```
using namespace std;

void FormatHandler::format(const uint8_t* rawdata, unsigned long len)
{
    vector<Data*>::iterator it;
    for (it = data_vector.begin(); it != data_vector.end(); ++it) {
        unsigned long required_len = (*it)->get_tot_size();
        if (required_len > len) {
            cout << "Not enough bytes to read Data. Returning" << endl;
            return;
        }
        memcpy((*it)->actual_data, rawdata, required_len);
        len -= required_len;
        rawdata += required_len;
    }
}
```

`Data` has no `actual_data` member

- There is an endianess problem with the *memcpy*;
- The memcpy might work with basic types ...
- definitely not with user-defined ones (e.g. the `DataImageRGB`);
- But there's a much bigger problem: this code does not compile ...

Data Vector formatting

DataExample3.cc

```
        e2.type = "float32";
        e2.name = "data_name_2";
        e2.len = 1;

        e3.type = "int8";
        e3.name = "data_name_3";
        e3.len = 4;

        e4.type = "bad_data_type";
        e4.name = "dontcare";
        e4.len = 1;

        out.push_back(e1);
        out.push_back(e2);
        out.push_back(e3);
        out.push_back(e4);

        return out;
}




class FormatHandler {
public:
        FormatHandler();
        ~FormatHandler();
        void format(uint8_t* rawdata, unsigned long len);
        inline const std::vector<Data*>& get_data_vector(void) const
```


Extensible design

Problem: the proposed design is not flexible at all

- Suppose we want to add a user-defined data-type (e.g. the RGB image):

```
<message_description>
   <data data_type="imageRGB" data_name="image_data" data_len="1"/>
</message_description>
```

- We need to:
  - derive the `Data` class in a new data type (e.g. `DataImageRGB`);
  - change the `FormatHandler` constructor (it has to create `DataImageRGB` when the `"imageRGB"` type-identifier is encountered);
  - change the `FormatHandler` formatting method to properly read bytes into the new `DataImageRGB` type

Extensible design

- The following modifications are required:
  - Adding a *DataImageRGB.h* and a *DataImageRGB.cpp* file
  - Editing the *FormatHandler.cpp* file
- The `FormatHandler` constructor and format method will explode

- Ideally we would like to limit the modifcations to the new file(s)
  - Adding a *DataImageRGB.h* and a *DataImageRGB.cpp* file

- **Solution**:
  - Delegate the construction of `DataImageRGB` to a *class factory*
  - Delegate the formatting operation to the `DataImageRGB` class itself

Delegated Data Vector formatting

Idea: each data object is capable to format itself

- each derived (concrete) data-class implements a specific format method
- `Data` has a *pure virtual* format method

```
class Data {
public:
    Data(const std::string& t, const std::string& n, unsigned long s, unsigned long l) :
        Metadata(t, s, l), name(n) {};
    inline std::string get_name(void) const { return name; };
    virtual unsigned long format_data(const uint8_t* buff, unsigned long len) = 0;
protected:
    std::string name;
};
```

- The `FormatHandler::format()` delegates the formatting to the `Data::format_data()`
- Notice that `Data::format_data()` returns the number of bytes used by the concrete `Data` to format itself.

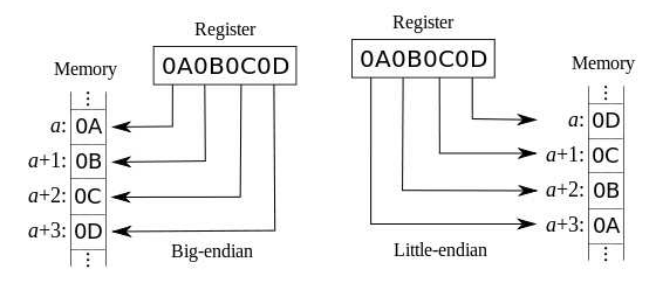
A simple excercise

Implement the “delegated” formatting for the concrete data classes.
`./examples/07.real-application_inheritance-examples/DataEx`

- Use the new defition of the class `Data`
- Implement the `format_data()` method for the concrete data classes
- Can you also fix the endianess issue?

The endianess issue

How the processor stores the 32-bit word `0x0A0B0C0D` at address a



- We don’t know the endianess of the source
- Endianess conversion might be necessary

- Data should also specify the endianess type

```
<message_description>
   <data data_type="uint16"  data_name="data_name_1" data_len="2" endianess="big-endian"/>
   <data data_type="float32" data_name="data_name_2" data_len="1" endianess="little-endian"/>
   <data data_type="int8"    data_name="data_name_3" data_len="4" endianess="little-endian"/>
</message_description>
```