

# 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

April 25, 2012

# Outline

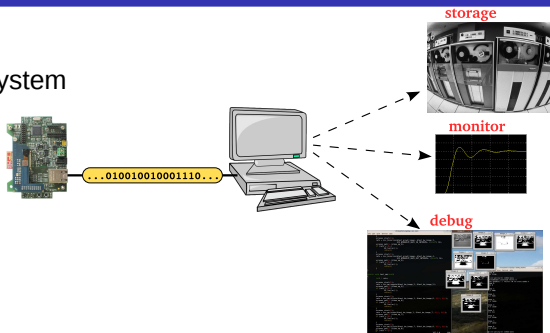
- 1 Previously...
- 2 Non-local variable initialization
- 3 Application to Factories
- 4 Initialization Order Fiasco
- 5 Construct On First Use idiom
- 6 Back to the duck-lab

# Outline

- 1 Previously...
- 2 Non-local variable initialization
- 3 Application to Factories
- 4 Initialization Order Fiasco
- 5 Construct On First Use idiom
- 6 Back to the duck-lab

# Data AcQquisition System

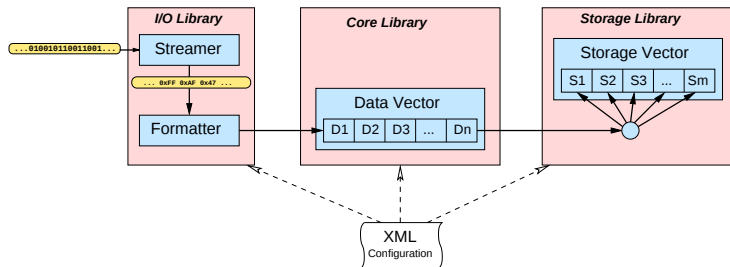
A basic DAQ system



*duck-lab*: an open-source DAQ framework

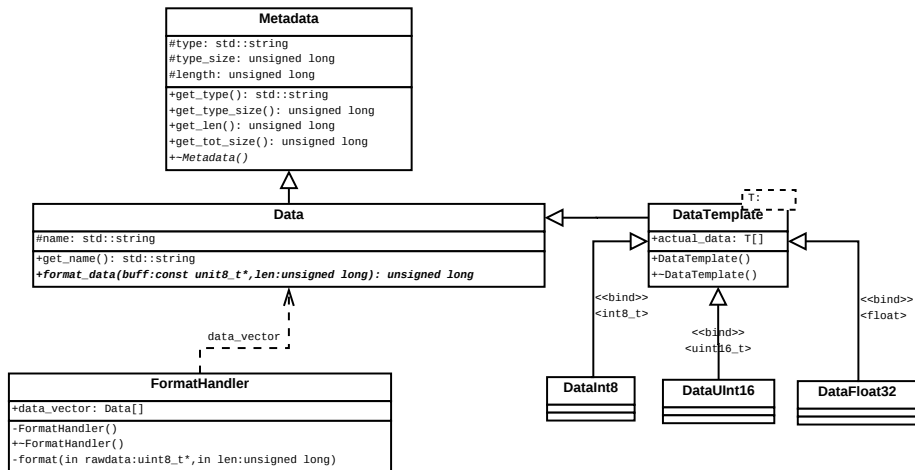
- Project (temporary) home page:  
<http://retis.sssup.it/~chris/duck-lab>
- Mercurial revision control: <https://rtn.sssup.it/hg/duck-lab>  
(hg clone <https://rtn.sssup.it/hg/duck-lab>)

# duck-lab architecture



- We have defined a hierarchy for Data
- We have defined the interaction with the Formatter, implemented by the `FormatHandler` class

# duck-lab architecture



# The FormatHandler class

- We have done Data creation/destruction
- We have done Data formatting

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

# The FormatHandler class

- We have done Data creation/destruction
- We have done Data formatting

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

Factory Method Pattern with string-based look-up table (`std::map`) (almost good)

Polymorphic destruction (good!)

Polymorphic formatting with delegation to the Data classes (good!)

# The FormatHandler class

- We have done Data creation/destruction
- We have done Data formatting

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

Factory Method Pattern with string-based look-up table (std::map) (almost good)

Polymorphic destruction (good!)

Polymorphic formatting with delegation to the Data classes (good!)

- So, given the following configuration (XML)

```
<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>
```

# FormatHandler ctor and dtor

DataExample5-to-exercise-solution/FormatHandler.cc

```
FormatHandler::FormatHandler()
{
    // We create the data vector according to XML config
    vector<XmlDataElement> config = parse_xml_data();

    vector<XmlDataElement>::const_iterator it;
    for (it = config.begin(); it != config.end(); ++it) {
        Data *d = DataFactory::create(it->type, it->name, it->len);
        if (!d) {
            cout << "Unable to create data type = '" << it->type <<
                "' ! Skipping" << endl;
            continue;
        }
        data_vector.push_back(d);
    }
}
```

DataExample5-to-exercise-solution/FormatHandler.cc

```
FormatHandler::~FormatHandler()
{
    // We have to destroy the data vector properly
    vector<Data*>::iterator it;
    for (it = data_vector.begin(); it != data_vector.end(); ++it)
        delete *it;
}
```

# DataFactory

Has anybody finished, if started, doing last exercise?

# DataFactory

Has anybody finished, if started, doing last exercise?

DataExample5-to-exercise-solution/DataFactory.h

```
class DataFactory {
public:
    typedef Data* (*creator_type_t)(const std::string&, unsigned long);

    static Data* create(const std::string& id, const std::string& name,
                        unsigned long len);
private:
    static std::map<std::string, creator_type_t> lookup_table;
};
```

# DataFactory

Has anybody finished, if started, doing last exercise?

DataExample5-to-exercise-solution/DataFactory.h

```
class DataFactory {
public:
    typedef Data* (*creator_type_t)(const std::string&, unsigned long);

    static Data* create(const std::string& id, const std::string& name,
                       unsigned long len);
private:
    static std::map<std::string, creator_type_t> lookup_table;
};
```

DataExample5-to-exercise-solution/DataFactory.cc

```
std::map<string, DataFactory::creator_type_t> DataFactory::lookup_table;

Data* DataFactory::create(const string& id, const string& n, unsigned long len)
{
    if (lookup_table.empty()) {
        lookup_table["int8"] = DataInt8::create;
        lookup_table["uint16"] = DataUInt16::create;
        lookup_table["float32"] = DataFloat32::create;
    }
    map<string, creator_type_t>::iterator elem = lookup_table.find(id);
    if (elem == lookup_table.end()) {
        cout << "Factory Creation Error: Creator not registered"
              << " for ID=" << id << endl;
        return NULL;
    }
    return elem->second(n, len);
}
```

# Outline

- 1 Previously...
- 2 Non-local variable initialization**
- 3 Application to Factories
- 4 Initialization Order Fiasco
- 5 Construct On First Use idiom
- 6 Back to the duck-lab

## *Static storage duration*

- Variables shall last for the entire duration of the program
- Variables with constructor/destructor having side effects cannot be eliminated

## *Static storage duration*

- Variables shall last for the entire duration of the program
- Variables with constructor/destructor having side effects cannot be eliminated

## *Non-local variables*

- Typically known as “global” variables
- Either with *static storage duration*...
- or *thread storage duration* (defined in C++11)

## *Static storage duration*

- Variables shall last for the entire duration of the program
- Variables with constructor/destructor having side effects cannot be eliminated

## *Non-local variables*

- Typically known as “global” variables
- Either with *static storage duration*...
- or *thread storage duration* (defined in C++11)

*“Non-local variables with static storage duration are initialized as a consequence of program initiation.”*

[C++11 standard: n3242, sec 3.6.2]

# Variables with static storage duration

## Three possible cases:

Non-local variables

```
int var1;  
static float var2;  
  
int main(void) { return 0; }
```

Local variables

```
void f(void)  
{  
    static int var = 123;  
    //...  
}
```

Static class data member  
(treated as non-local  
variable)

```
class Test {  
    //...  
    static int var;  
};  
  
int Test::var;
```

# Variables with static storage duration

## Three possible cases:

Non-local variables

```
int var1;  
static float var2;  
  
int main(void) { return 0; }
```

Static as linkage rule

Local variables

```
void f(void)  
{  
    static int var = 123;  
    //...  
}
```

Declaration of static duration

Static class data member  
(treated as non-local  
variable)

```
class Test {  
    //...  
    static int var;  
};  
  
int Test::var;
```

Declare static class member

**Necessary** to define (allocate)  
data member

Notice the different usages of static

# Initialization Type

Two possible initializations

# Initialization Type

## Two possible initializations

- *static initialization*

### zero initialization

```
int var;  
int* ptr;  
int array[3];  
float real;  
int main(void) {return 0;}
```

### constant initialization

```
int var = 123;  
int* ptr = &var1;  
int array[3] = {1, 2, 3};  
int main(void) {return 0;}
```

# Initialization Type

## Two possible initializations

- *static initialization*

### zero initialization

```
int var;  
int* ptr;  
int array[3];  
float real;  
int main(void) {return 0;}
```

### constant initialization

```
int var = 123;  
int* ptr = &var1;  
int array[3] = {1, 2, 3};  
int main(void) {return 0;}
```

- *dynamic initialization*: all other initializations

```
inline int f() {return 123;}  
int var = f();  
float real = 3.14; // Dynamic Initialization!  
int main(void) {return 0;}
```

# Initialization: Static vs Dynamic

Considering non-local variable with static storage duration

## Static initialization

- Initialization value computed at compile time
- All static initializations occur before any dynamic initialization

# Initialization: Static vs Dynamic

Considering non-local variable with static storage duration

## Static initialization

- Initialization value computed at compile time
- All static initializations occur before any dynamic initialization

## Dynamic initialization

- Initialization value computed at runtime

# Initialization: Static vs Dynamic

Considering non-local variable with static storage duration

## Static initialization

- Initialization value computed at compile time
- All static initializations occur before any dynamic initialization

## Dynamic initialization

- Initialization value computed at runtime
- Under certain circumstances static initialization **might** be used in place of dynamic one

# Initialization: Static vs Dynamic

## Considering non-local variable with static storage duration

### Static initialization

- Initialization value computed at compile time
- All static initializations occur before any dynamic initialization

### Dynamic initialization

- Initialization value computed at runtime
- Under certain circumstances static initialization **might** be used in place of dynamic one
- Initialization **might** occur before first statement of `main`, otherwise...
  - *“it shall occur before the first odr-use of any function or variable defined in the same translation unit as the variable to be initialized.”* [C++11 standard: n3242, sec 3.6.2, par 4]
  - *“A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used”* [C++11 standard: n3242, sec 3.6.2, note 34]

# Initialization: Static vs Dynamic

## Considering non-local variable with static storage duration

### Static initialization

- Initialization value computed at compile time
- All static initializations occur before any dynamic initialization

### Dynamic initialization

- Initialization value computed at runtime
- Under certain circumstances static initialization **might** be used in place of dynamic one
- Initialization **might** occur before first statement of `main`, otherwise...
  - *“it shall occur before the first odr-use of any function or variable defined in the same translation unit as the variable to be initialized.”* [C++11 standard: n3242, sec 3.6.2, par 4]
  - *“A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used”* [C++11 standard: n3242, sec 3.6.2, note 34]

### Note:

- Variables might be static- (zero-) initialized first and dynamic-initialized later.
- User-defined types variable (classes) are subject to static and dynamic initialization too.

# User-defined type: static initialization

*“if  $T$  is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized and padding is initialized to zero bits”*

*[C++11 standard: n3242, sec 8.5, par 5]*

# User-defined type: static initialization

*"if T is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized and padding is initialized to zero bits"*

[C++11 standard: n3242, sec 8.5, par 5]

```
struct Simple {
    int var;
    int* ptr;
    char str[10];
};

Simple s;

int main(void)
{
    cout << "Simple has: " << s.var << " " <<
        s.ptr << " '" << s.str << "'" << endl;
    return 0;
}
```

# User-defined type: static initialization

*"if T is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized and padding is initialized to zero bits"*

[C++11 standard: n3242, sec 8.5, par 5]

```
struct Simple {
    int var;
    int* ptr;
    char str[10];
};

Simple s;

int main(void)
{
    cout << "Simple has: " << s.var << " " <<
        s.ptr << " '" << s.str << "'" << endl;
    return 0;
}
```

```
Simple has: 0 0 ''
```

# User-defined type: dynamic initialization

When zero-initialization construction is not sufficient...

```
struct Simple {  
    int var;  
    int* ptr;  
    char str[10];  
    Simple() : var(123), ptr(&var)  
    {  
        strcpy(str, "Hello");  
    }  
};  
  
Simple s;  
  
int main(void)  
{  
    cout << "Simple has: " << s.var << " " <<  
        s.ptr << " '" << s.str << "'" << endl;  
    return 0;  
}
```

# User-defined type: dynamic initialization

When zero-initialization construction is not sufficient...

```
struct Simple {  
    int var;  
    int* ptr;  
    char str[10];  
    Simple() : var(123), ptr(&var)   
    {  
        strcpy(str, "Hello");  
    }  
};  
  
Simple s;  
  
int main(void)  
{  
    cout << "Simple has: " << s.var << " " <<  
        s.ptr << " '" << s.str << "'" << endl;  
    return 0;  
}
```

User-provided constructor

Requesting a **run-time** call to the user-provided constructor

# User-defined type: dynamic initialization

When zero-initialization construction is not sufficient...

```
struct Simple {  
    int var;  
    int* ptr;  
    char str[10];  
    Simple() : var(123), ptr(&var)  
    {  
        strcpy(str, "Hello");  
    }  
};  
  
Simple s;  
  
int main(void)  
{  
    cout << "Simple has: " << s.var << " " <<  
        s.ptr << " '" << s.str << "'" << endl;  
    return 0;  
}
```

User-provided constructor

Requesting a **run-time** call to the user-provided constructor

```
Simple has: 123 0x6013a0 'Hello'
```

# And now?

- The question is: why do we care about all this stuff?

# And now?

- The question is: why do we care about all this stuff?
- It turns out that non-local variable initialization can be exploited to execute code “at start-up”.

# And now?

- The question is: why do we care about all this stuff?
- It turns out that non-local variable initialization can be exploited to execute code “at start-up”.

Dynamic initialization **might** occur before first statement of `main`, otherwise...

- *“it shall occur before the first odr-use of any function or variable defined in the same translation unit as the variable to be initialized.”*
- *“A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used”*

Suppose to have:

- a non-local variable of user-defined type (class)
- which a user-provided constructor
- which has side-effects (produces changes in the execution environment)

# Example

```
struct MyClass {
    MyClass()
    {
        cout << "Make world..." << endl;
        // ...
    }
    //...
};

MyClass c;

int main(void)
{
    cout << "Hello world!" << endl;
    return 0;
}
```

# Example

```
struct MyClass {  
    MyClass()  
    {  
        cout << "Make world..." << endl;  
        // ...  
    }  
    //...  
};  
  
MyClass c;  
  
int main(void)  
{  
    cout << "Hello world!" << endl;  
    return 0;  
}
```

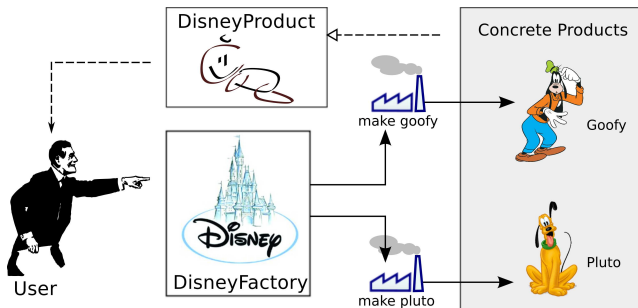
```
Make world...  
Hello world!
```

# Outline

- 1 Previously...
- 2 Non-local variable initialization
- 3 Application to Factories**
- 4 Initialization Order Fiasco
- 5 Construct On First Use idiom
- 6 Back to the duck-lab

# The Disney's cartoons problem

- The *user* wants to animate cartoons
- The *user* does not care about their creation process



# C++ implementation

FactoryExample4.h

```
class DisneyProduct {  
public:  
    virtual ~DisneyProduct() {};  
    inline std::string get_name(void) const { return name; };  
protected:  
    DisneyProduct(const std::string& n) : name(n) {};  
private:  
    DisneyProduct(const DisneyProduct&);  
    DisneyProduct &operator=(const DisneyProduct&);  
    std::string name;  
};
```

FactoryExample4.h

```
class Goofy : public DisneyProduct {  
private:  
    Goofy() : DisneyProduct("goofy") {};  
    static DisneyProduct* create(void) { return new Goofy(); };  
    friend class DisneyFactory;  
};  
  
class Pluto : public DisneyProduct {  
private:  
    Pluto() : DisneyProduct("pluto") {};  
    static DisneyProduct* create(void) { return new Pluto(); };  
    friend class DisneyFactory;  
};
```

# C++ implementation

FactoryExample4.h

```
class DisneyUser {  
public:  
    DisneyUser();  
    ~DisneyUser();  
    void animate(void);  
private:  
    std::vector<DisneyProduct*> toons;  
};
```

# C++ implementation

FactoryExample4.cc

```
DisneyUser::DisneyUser()
{
    const char* config[] = {"goofy", "pluto", "pluto", "invalid_id"};

    for (size_t i = 0; i < 4; i++) {
        DisneyProduct *p = DisneyFactory::create(config[i]);
        if (p)
            toons.push_back(p);
    }
}

DisneyUser::~DisneyUser()
{
    vector<DisneyProduct*>::iterator it;
    for (it = toons.begin(); it != toons.end(); ++it)
        delete *it;
}

void DisneyUser::animate(void)
{
    vector<DisneyProduct*>::const_iterator it;
    for (it = toons.begin(); it != toons.end(); ++it)
        cout << "Hullo, my name is " << (*it)->get_name() << endl;
}
```

# C++ implementation

FactoryExample4.h

```
class DisneyFactory {  
public:  
    static DisneyProduct* create(const std::string& id);  
private:  
    typedef DisneyProduct* (*creator_type_t)(void);  
    static std::map<std::string, creator_type_t> lookup_table;  
};
```

FactoryExample4.cc

```
map<std::string, DisneyFactory::creator_type_t> DisneyFactory::lookup_table;  
  
DisneyProduct* DisneyFactory::create(const string& id)  
{  
    if (lookup_table.empty()) {  
        lookup_table["goofy"] = Goofy::create;  
        lookup_table["pluto"] = Pluto::create;  
    }  
    if (lookup_table.find(id) == lookup_table.end())  
        return NULL;  
    return lookup_table[id]();  
}
```

# Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:

# Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:
  - 1 definition of the new `DisneyProduct` (e.g. `DonaldDuck`)

# Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:
  - 1 definition of the new `DisneyProduct` (e.g. `DonaldDuck`)
  - 2 adding a new entry in the look-up table...

```
if (lookup_table.empty()) {  
    lookup_table["goofy"] = Goofy::create;  
    lookup_table["pluto"] = Pluto::create;  
    lookup_table["donal duck"] = DonaldDuck::create;  
}
```

# Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:
  - 1 definition of the new `DisneyProduct` (e.g. `DonaldDuck`)
  - 2 adding a new entry in the look-up table...

```
if (lookup_table.empty()) {  
    lookup_table["goofy"] = Goofy::create;  
    lookup_table["pluto"] = Pluto::create;  
    lookup_table["donal duck"] = DonaldDuck::create;  
}
```

... in the `DisneyFactory` class implementation

# Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:
  - 1 definition of the new `DisneyProduct` (e.g. `DonaldDuck`)
  - 2 adding a new entry in the look-up table...

```
if (lookup_table.empty()) {  
    lookup_table["goofy"] = Goofy::create;  
    lookup_table["pluto"] = Pluto::create;  
    lookup_table["donal duck"] = DonaldDuck::create;  
}
```

... in the `DisneyFactory` class implementation

- Using non-local variable initialization we can avoid step 2!

# Factory with automatic registration

- The `DisneyFactory` shall contain the logic to register the creation method in the look-up table.
- This registration is performed at run-time and at start-up

# Factory with automatic registration

- The `DisneyFactory` shall contain the logic to register the creation method in the look-up table.
- This registration is performed at run-time and at start-up
- But where do we write this code?

# Factory with automatic registration

- The `DisneyFactory` shall contain the logic to register the creation method in the look-up table.
- This registration is performed at run-time and at start-up
- But where do we write this code?
- In the constructor, of course!

FactoryExample5.h

```
class DisneyFactory {  
public:  
    typedef DisneyProduct* (*creator_type_t)(void);  
  
    DisneyFactory(const std::string& id, creator_type_t func);  
    static DisneyProduct* create(const std::string& id);  
private:  
    static std::map<std::string, creator_type_t> lookup_table;  
};
```

# Factory with automatic registration

FactoryExample5.cc

```
map<std::string, DisneyFactory::creator_type_t> DisneyFactory::lookup_table;

DisneyFactory::DisneyFactory(const string& id, creator_type_t func)
{
    cout << "Factory: registering creator for id=" << id << endl;
    if (lookup_table.find(id) != lookup_table.end()) {
        cout << "Error: ID already in look-up table" << endl;
        return;
    }
    lookup_table[id] = func;
}

DisneyProduct* DisneyFactory::create(const string& id)
{
    if (lookup_table.find(id) == lookup_table.end())
        return NULL;
    return lookup_table[id]();
}
```

# Factory with automatic registration

- Each concrete `DisneyProduct` class shall automatically register itself...
- just defining a non-local variable with static storage duration of type `DisneyFactory`.

```
class Goofy : public DisneyProduct {  
public:  
    static DisneyProduct* create(void) { return new Goofy(); };  
private:  
    Goofy() : DisneyProduct("goofy") {};  
};  
  
// In the implementation file...  
  
DisneyFactory register_goofy("goofy", Goofy::create);
```

The full example is:

`./examples/10.real-application_moreOnCreation-examples/Fact`

# Outline

- 1 Previously...
- 2 Non-local variable initialization
- 3 Application to Factories
- 4 Initialization Order Fiasco**
- 5 Construct On First Use idiom
- 6 Back to the duck-lab

*“Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit.”*

*[C++11 standard: n3242, sec 3.6.2, par 2]*

In other words:

- global variables are initialized following the order of appearance in the source file;
- but the initialization order is not guaranteed between variables of different source files!

# Initialization order fiasco

## Goofy.cc

```
class Goofy : public DisneyProduct {  
public:  
    static DisneyProduct* create(void) { return new Goofy(); };  
private:  
    Goofy() : DisneyProduct("goofy") {};  
};  
  
DisneyFactory reg("goofy", Goofy::create);
```

## DisneyFactory.cc

```
map<std::string, DisneyFactory::creator_type_t>  
    DisneyFactory::lookup_table;  
  
DisneyFactory::DisneyFactory(const string& id, creator_type_t func)  
{  
    if (lookup_table.find(id) != lookup_table.end()) {  
        cout << "Error: already in look-up table" << endl;  
        return;  
    }  
    lookup_table[id] = func;  
}
```

# How to prevent the fiasco

The problem of previous example:

- If `DisneyFactory::lookup_table` is not initialized before the non-local object reg ...
- then the call to the constructor (registration mechanism) `DisneyFactory::DisneyFactory( . . . )` fails!

# How to prevent the fiasco

The problem of previous example:

- If `DisneyFactory::lookup_table` is not initialized before the non-local object reg ...
- then the call to the constructor (registration mechanism) `DisneyFactory::DisneyFactory( ... )` fails!

Solution for the previous example?

# How to prevent the fiasco

The problem of previous example:

- If `DisneyFactory::lookup_table` is not initialized before the non-local object `reg ...`
- then the call to the constructor (registration mechanism) `DisneyFactory::DisneyFactory( ... )` fails!

Solution for the previous example?

- Make `DisneyFactory::lookup_table` a pointer to map rather than a map object.
- As a pointer it shall be zero-initialized (static initialization) before any dynamic initialization.
- Delay the creation of the actual map until it is necessary: when it is used by `DisneyFactory::DisneyFactory( ... )`.

# How to prevent the fiasco

## DisneyFactory.cc

```
map<std::string, DisneyFactory::creator_type_t>*
    DisneyFactory::lookup_table;

DisneyFactory::DisneyFactory(const string& id, creator_type_t func)
{
    if (!lookup_table)
        lookup_table = new map<std::string, creator_type_t>();
    if (lookup_table->find(id) != lookup_table->end()) {
        cout << "Error: already in look-up table" << endl;
        return;
    }
    (*lookup_table)[id] = func;
}
```

# How to prevent the fiasco

## DisneyFactory.cc

```
map<std::string, DisneyFactory::creator_type_t>*
    DisneyFactory::lookup_table;

DisneyFactory::DisneyFactory(const string& id, creator_type_t func)
{
    if (!lookup_table)
        lookup_table = new map<std::string, creator_type_t>();
    if (lookup_table->find(id) != lookup_table->end()) {
        cout << "Error: already in look-up table" << endl;
        return;
    }
    (*lookup_table)[id] = func;
}
```

The generalization of this approach is known as “*Construct On First Use idiom*”.

# Outline

- 1 Previously...
- 2 Non-local variable initialization
- 3 Application to Factories
- 4 Initialization Order Fiasco
- 5 Construct On First Use idiom**
- 6 Back to the duck-lab

# Construct On First Use Idiom

## Objective:

- Ensure that an object is initialized before its first use.
- Specifically, ensure that a non-local static object is initialized before its first use.

## Solution:

- Encapsulate the creation of the “critical” object in a function.
- Basically the object is now a **local** variable with static storage duration which is initialized **only** when the function is called!
- Also known as lazy initialization/evaluation (not at startup)

```
MyClass& getMyObj()  
{  
    static MyClass* p = new MyClass;  
    return *p;  
}
```

# Example [<http://www.parashift.com/c++-faq-lite/ctors.html>]

x.cc

```
#include "Fred.h"  
Fred x;
```

y.cc

```
#include "Barney.h"  
Barney y;
```

Barney.cc

```
#include "Barney.h"  
Barney::Barney()  
{  
    //...  
    x.goBowling();  
    //...  
}
```

X.CC

```
#include "Fred.h"
Fred x;
```

y.CC

```
#include "Barney.h"
Barney y;
```

Barney.cc

```
#include "Barney.h"
Barney::Barney()
{
    //...
    x.goBowling();
    //...
}
```

X.CC

```
#include "Fred.h"

Fred& x()
{
    static Fred* ans = new Fred();
    return *ans;
}
```

Barney.cc

```
#include "Barney.h"

Barney::Barney()
{
    //...
    x().goBowling();
    //...
}
```

# Alternative implementation...

## Dynamic allocation

```
#include "Fred.h"

Fred& x()
{
    static Fred* ans = new Fred();
    return *ans;
}
```

- The destructor for ans is not called, ever!
- Not good if destructor is non-trivial.

## Local static

```
#include "Fred.h"

Fred& x()
{
    static Fred ans;
    return ans;
}
```

- The destructor for ans is called on program termination.
- Might generate destruction order fiasco!

# Outline

- 1 Previously...
- 2 Non-local variable initialization
- 3 Application to Factories
- 4 Initialization Order Fiasco
- 5 Construct On First Use idiom
- 6 Back to the duck-lab**

# DataFactory with automatic registration

- The DataFactory class shall have a constructor to register the creator functions.
- The creator functions shall be defined by each concrete data-class:
  - DataInt8
  - DataUInt16
  - DataFloat32
  - ...
- A utility macro might be used to generate the code for the creator function given a public constructor.

The full example is:

```
./examples/10.real-application_moreOnCreation-examples/Data
```

# Consideration

- What about *extensibility*?
- Adding new Data requires the following modifications:

# Consideration

- What about *extensibility*?
- Adding new Data requires the following modifications:
  - 1 definition of the new concrete Data  
(e.g. DataFloat64, DataImageRGB)

- What about *extensibility*?
- Adding new Data requires the following modifications:
  - 1 definition of the new concrete Data (e.g. DataFloat64, DataImageRGB)
  - 2 adding the automatic registration (one line if using the macro)

# Consideration

- What about *extensibility*?
- Adding new Data requires the following modifications:
  - 1 definition of the new concrete Data (e.g. DataFloat64, DataImageRGB)
  - 2 adding the automatic registration (one line if using the macro)

Notice:

- Both changes are to the implementation file of the new class!
- No more changes are required to other files!

# Different creation parameters

- We have now an extensible solution with respect to new Data.

# Different creation parameters

- We have now an extensible solution with respect to new Data.
- However, the creation function accepts always the same parameters!

DataExample6/DataFactory.h

```
class DataFactory {  
public:  
    typedef Data* (*creator_type_t)(const std::string&, unsigned long);  
    DataFactory(const std::string& id, creator_type_t func);
```

- Fine for basic data types...

DataExample6/DataInt8.h

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

# Different creation parameters

- Consider the DataImageRGB example...
- a new concrete data to manage images in RGB format

```
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...
    };

    unsigned long format_data(const uint8_t* buff, unsigned long len)
    {
        // Formatting...
    };
    // ...
};
```

- The constructor requires two more params: width and height

# Different creation parameters

The parameters width and height should be coded in the XML configuration file

```
<message_description>  
  <data data_type="imageRGB" data_name="data_name_3" data_len="1">  
    <RGB_config width="160" height="120">  
  </data>  
</message_description>
```

# Different creation parameters

The parameters width and height should be coded in the XML configuration file

```
<message_description>
  <data data_type="imageRGB" data_name="data_name_3" data_len="1">
    <RGB_config width="160" height="120">
  </data>
</message_description>
```

**Problem:** is the configuration logic: [DataExample5-to-exercise-solution/FormatHandler.cc](#)

```
FormatHandler::FormatHandler()
{
    // We create the data vector according to XML config
    vector<XmlDataElement> config = parse_xml_data();

    vector<XmlDataElement>::const_iterator it;
    for (it = config.begin(); it != config.end(); ++it) {
        Data *d = DataFactory::create(it->type, it->name, it->len);
        if (!d) {
            cout << "Unable to create data type = '" << it->type <<
                "' ! Skipping" << endl;
            continue;
        }
        data_vector.push_back(d);
    }
}
```

# Different creation parameters

The parameters width and height should be coded in the XML configuration file

```
<message_description>
  <data data_type="imageRGB" data_name="data_name_3" data_len="1">
    <RGB_config width="160" height="120">
  </data>
</message_description>
```

**Problem:** is the configuration logic: [DataExample5-to-exercise-solution/FormatHandler.cc](#)

```
FormatHandler::FormatHandler()
{
    // We create the data vector according to XML config
    vector<XmlDataElement> config = parse_xml_data();

    vector<XmlDataElement>::const_iterator it;
    for (it = config.begin(); it != config.end(); ++it) {
        Data *d = DataFactory::create(it->type, it->name, it->len);
        if (!d) {
            cout << "Unable to create data type = '" << it->type <<
                "' ! Skipping" << endl;
            continue;
        }
        data_vector.push_back(d);
    }
}
```

**Solution:**

- Delegate the XML parsing to the Factory...
- and to enforce extensibility, eventually delegate to the concrete data class!

# duck-lab approach

The solution adopted in the *duck-lab* uses a “two-way factory”:

- A Parser class is defined to access the XML
- A DataConfig class is defined to store the config information retrieved through the Parser
- The Data class is created from the DataConfig

Motivation: separating the concrete Data class from its configuration.

