Dynamization of C++ Static Libraries

Nicolas Pouillard **and** Damien Thivolle

Technical Report n^o0602, January 30, 2006

Static C++ allows designers to develop efficient and generic libraries, but for the end user such libraries are very restricting. Indeed compilation cycles are so long that it forbids prototyping. To overcome this shortcoming, wrappers generators such as SWIG allow to pre-instantiate static classes and functions and then make them available in a higher-level language.

In our opinion, such approaches have drawbacks. They force the end user to learn a new language to use a C++ library and they can not use classes or functions if they are not available yet. From users feedback, what is really needed is a way to use static C++ from within a C++ dynamic environment and without facing deadly compilation times.

To respond to that need, we developed a C++ environment that allows static C++ functions and classes manipulation. We use just in time compilation with a cache system to compile classes and functions on demand. Using advanced C++ programming techniques, we manage to rend the usage of our environment very handy for the end user, thus allowing fast and efficient prototyping.

Keywords
C++, static, dynamic, JIT, compilation cache

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22

Contents

1 Introduction **4**

1.1 Static/Dynamic Bridge 4

1.2 Just in Time Compilation 4

1.3 Related work 5

1.4 Our approach 5

2 On the user side **6**

2.1 Transparent method and function calls 6

2.1.1 Motivations 6

2.1.2 Function calls 6

2.1.3 Method calls 7

2.2 User interface 7

3 Technical Specifications **10**

3.1 External polymorphism pattern 10

3.2 Call dependent wrappers 11

3.2.1 Why not generate function dependent wrappers? 11

3.2.2 The arguments' type 12

3.3 Functions return type 12

3.4 JIT module 15

3.4.1 The C++ generation unit 16

3.4.2 The compilation unit 16

3.4.3 The dynamic library loading, function pointer fetching, casting and calling 16

4 Conclusion **17**

4.0.4 Summary 17

4.0.5 Future work 17

4.1 Acknowledgments 18

5 Bibliography **19**

Chapter 1

Introduction

1.1 Static/Dynamic Bridge

Static C++ provides programmers with techniques allowing to develop efficient and generic libraries. The two main drawbacks are: the compilation cycles are very long (it often takes more than a minute) and such libraries can not really be used in dynamic environment.

A static/dynamic bridge aims at making features from a static library available in a dynamic environment. There exist tools such as SWIG which generate such bridges allowing parts of a static library to be used from within a dynamic language like Ruby or Python . As far as we know, a static/dynamic bridge can work in two different ways:

- Using explicit instantations of classes and functions of a static library, it is possible to wrap them so that they become a module in a dynamic language. Such a method would only need one compilation but it wouldn't be possible to use types or functions that were not compiled.
- The just in time compilation technique allows to compile classes or functions when and only when they are needed. This makes it possible to bind an entire static library in a dynamic environment.

1.2 Just in Time Compilation

In order to provide a dynamic mechanism with a static language we must call the C++ compiler at run-time. Especially with this language, compilation costs are quite heavy thus a *lazy* system and a *cache* are used. They will be described in Cache handling.

Some design choices for generated wrappers:

- They must be non-dependent upon the return type since the C++ overloading does not take it in account. See the section Functions return type for more details about this problem.
- Wrappers are call dependent, not function dependent (see Call dependent wrappers), so the generation must produce the same wrapper for the same parameters:
  - the fully qualified name of the function.
  - the type of each argument.
  - the name of every header needed to find this function **definition**.
  - the policy allocation of arguments will also be taken in account.

1.3 Related work

We based our work on Alexandre Duret-Lutz 's research from 2001 in which he described how to use the external polymorphism pattern to use Olena (a static library for image processing) from a graphical interface (i.e. a dynamic environment). He suggested to use *JIT* compilation in order to generate for each instance of a function a wrapping function taking a *component* as an argument. That component contained the actual function arguments (wrapper within dynamic proxies) and had all the information needed to downcast those arguments from their proxy to their real type.

Another work worth to be mentionned is the attempt Loic Fosse made to patch SWIG (a wrapper generator) so that it could handle statically typed functions. As we aimed at providing a C++ solution to the problem, we did not really use his work since SWIG generates wrappers for dynamic languages such as Ruby or Python .

1.4 Our approach

We aimed at providing a dynamic C++ environment for using static C++ classes and functions. In that environment, it is possible to call functions and instantiate objects and then call the methods provided by those objects. Type conversions are done automatically and return values are handled correctly so that everything is as intuitive and transparent as possible for the end user.

This report is a legacy for both the future maintainers and users. It is organized as follows. The first part is a sort of user documentation, it shows how to use our work, many examples will guide the readers. The second part is more technical, it explains how we addressed the main issues we faced and how the whole thing works.

Chapter 2

On the user side

This part deals with the model and the discuss can also stand as a User Documentation.

To have a system as simpler as possible we decided to be very close to the traditional way to call functions and methods.

2.1 Transparent method and function calls

Our motivations for a very transparent way to make function and method calls were mainly influenced by the user point of view.

2.1.1 Motivations

We want to stay very close to the classical way in order to make the transition to dynamic version very easy, but also to make it easily learnable without any knowledge about some complex `C++` techniques.

2.1.2 Function calls

Dealing with functions is the main part of this work, other things like methods, operators or constructors are just a variant of classical functions. So as a first task we search a way to express dynamic function calls. For instance it will be useful to simply have a sort of special namespace (call it `dyn`) containing the dynamic wrappers for each dynamized function.

For example this call:

```
a_type result = foo(some, arguments, to, the, foo, function);
```

Can be turned into this one:

```
a_type result = dyn::foo(some, arguments, to, the, foo, function);
```

Or this one (if the type is to complex to type):

```
var result = dyn::foo(some, arguments, to, the, foo, function);
```

2.1.3 Method calls

In a same way we want to keep the use simplicity, but methods are slightly different, so to make every one happy we propose two ways to call a method on an object. The first one is like a function call but with the object as first argument. The second one behaves like a classical method call but is a little more complex to setup.

Example:

```
meth foo("foo");  // declare the foo method.
aClass anObject(...); // a new object.
// Way 1
foo(anObject, some, others, arguments);
// Way 2
DYN_REGISTER_METHOD(foo);
anObject.foo(some, others, arguments);
// Operators are already defined
var x = "test.";
std::cout « x « std::endl; // output: test.
x[4] = '!';
std::cout « x « std::endl; // output: test!
```

2.2 User interface

This part describes how to use our work and what we provide to use static functions or classes in a dynamic C++ environment.

As our work was designed to use static C++ libraries, the following explanations will always refer to the same static library in our example. This library is a matrix library using the following interface:

```
// This is the file matrix.hh
template <unsigned Dim>
struct Matrix
{
```

```
  Matrix(std::istream);
  Matrix(Matrix<Dim>);
  Matrix<Dim> operator*(const Matrix<Dim> rhs) const;
  Matrix<Dim> operator+(const Matrix<Dim> rhs) const;
};
template <unsigned Dim>
Matrix<Dim> mult(const Matrix<Dim> lhs, const Matrix<Dim> rhs);
{
  return lhs * rhs;
}
template <unsigned Dim>
Matrix<Dim> add(const Matrix<Dim> lhs, const Matrix<Dim> rhs)
{
  return lhs + rhs;
}
#include <matrix.hxx> // Implementations
```

To use such a library, the user needs to indicate where the file ‘matrix.hh’ is located. The *JIT* module might need additionnal information such as the compiler flags and the link editor flags:

```
 dyn::include_dir(PATH_TO_MATRIX_HH); // Adds a directory to the include path
dyn::include("matrix.hh"); // Tells the JIT module which files need to be included
dyn::cflags("-O -W -Wall -Werror"); // Sets the flags to be used by the compiler
dyn::ldflags("-lm"); // Sets the flags to be used by the link editor
```

Now that everything has been correctly set for the *JIT* module, we can start using the library. First we need to instantiate two matrices. To do so, a *ctor* (constructor) needs to be used. In our environment, static objects are of the same type: *data* (*var* is a typedef on *data*), therefore the matrices we instantiate are of the type *data*.

```
using namespace dyn::language; // makes available some types fun. var, val...
ctor mk_matrix_2d("matrix<2>"); // Constructor for 2D matrices
var mat_a = mk_matrix_2d(std::cin);
var mat_b = mk_matrix_2d(std::cin);
```

If one of the matrices was meant to be constant, the correct way to express it, would be to declare it of type *val* which means value. A value is just a *data* that can not be assigned:

```
val const_mat = mk_matrix_2d(std::cin);
```

A function needs to be instantiated before it can be called. In the dynamic environment, a function is represented by a *functor* so that the calls remain intuitive:

```
fun mult("mult");
fun add("add");
var mat_c = mult(mat_a, add(mat_b, mat_a)); // c = a * (a + b)
```

A method also needs to be instantiated, except for the operators for which we automatically create wrappers even if they do not exist. One can try to call an operator as a method of an object but if this operator doesn't exist in the object class, the call will fail.

```
var mat_d = a * b + c; // Operators are automatically created.
```

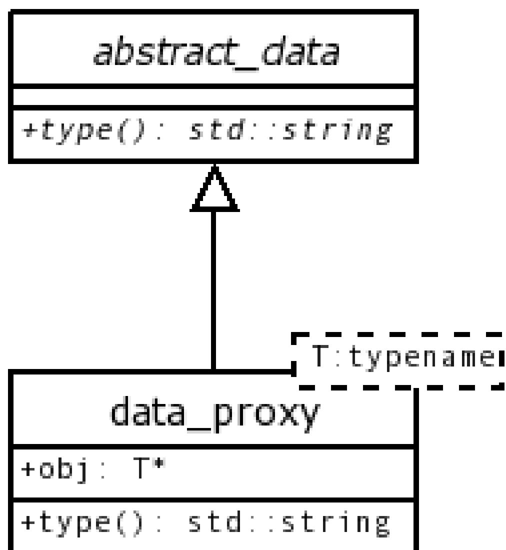
As showed above, every class and every function from a static library can be used within our environment. To achieve this, we had to address many issues using advanced C++ tricks that will be presented in the next section.

Chapter 3

Technical Specifications

3.1 External polymorphism pattern

The external polymorphism pattern is a *design pattern* that consists of wrapping different classes under a single upper class so that they can be manipulated in a same way.



The *data_proxy* class has a template parameter that holds the type of the class it wraps. A *data_proxy* class inherits from a non-template abstract class called *abstract_data* which only provides a *type* method to retrieve the type of the actual wrapped class.

We used this design pattern to only manipulate objects of type *abstract_data* in our dynamic environment. The *type()* method is used by the *JIT* (Compilation Just in Time) module to generate the wrapping function. We will explain the JIT module more thoroughly later. The

string corresponding to the type of an object is retrieved using the library liberty to *demangle* the C++ function names returned by *typeid*.

This *abstract_data* is used to pass on the function arguments from the dynamic environment to the static environment and to retrieve in the dynamic environment the value returned by a function in the static environment. Therefore we instantiate *abstract_data* objects and to make it easier for the end user, we provide a *data* class that aggregates a *abstract_data* and that can be instantiated on every kind of objetcs. It handles automatic conversions (if they are possible) between objects so that the user can use a *data* object wrapping an object of type *foo* as if it was a real instance of *foo*.

3.2 Call dependent wrappers

3.2.1 Why not generate function dependent wrappers?

It seems appealing to generate only one wrapper by function but as one knows the C++ overloading resolution rules are complex and make our work behave exactly like it should will be a pain to develop/maintain.

Conversions must also be taken in account because automatic conversions can happen when conversion operators are used.

Moreover to access to all this prototype information one must either extract this form sources or maintain a database with all functions' prototype of the library.

To this list we can add optional arguments, template functions and so on…

Then in front of all these problems we chose to make a wrapper by call. Of course the cache system (see Cache handling) tends to be equal to the "one wrapper by function" method when overloading, conversions, and optional arguments are not used.

Parameters of the wrapper generation: The wrapper generation is parametrized by different things. It's important to well describe all of them because it helps to understand why two calls have the same cache entry or not.

The fully qualified function name: It seems obvious to require the name of the function in order to compile its wrapper. Here we just make it clear that the name required must be non ambiguous, so all namespaces must be well specified.

The headers: At least one header file name is needed to find the function definition. If you have an overloaded function you need to supply all headers where the function is to use it in the dynamic side with its different types.

3.2.2 The arguments' type

In order to generate the good wrapper, our library must know the type of each argument of the call (but is not dependent upon the return type of the function).

Arguments allocation policy: This requirement can be surprising and will be perhaps removed in a future work but for now it was easier to consider this policy inside the argument's type.

Example:
```
// call_dependent_wrappers_example.hh
namespace foo {
  void bar(long i, std::ostream o, std::string j)
  { o « "foo::bar(" « i « ", \"" « j « "\")" « std::endl; }
}
// call_dependent_wrappers_example.cc
#include <iostream>
#include <dyn-all.hh>
using namespace dyn::language;
int main()
{
  dyn::include_dir(get_current_dir_name());
  dyn::fun dyn_foo_bar("foo::bar", "call_dependent_wrappers_example.hh");
  int i = 42;
  val j = "test";
  dyn_foo_bar(i, std::cout, j);
}
```

In this code snippet the parameters are:

- The name: foo::bar
- The header: call_dependent_wrappers_example.hh
- The arguments' type:
  - int (not long) and in fact it's dyn::data_proxy_by_ref because argument allocation policy is used.
  - dyn::data_proxy_by_ref > > (not std::ostream)
  - dyn::data_proxy_by_ref (not std::string)

3.3 Functions return type

Knowledge about the functions return type was needed in order to choose the right allocation policy for the data_proxies.

First of all, we need to introduce a property of the *sizeof* operator. When calling *sizeof* on a function call, it returns the size of the return type of the function. The most important thing is that the function is not called (meaning that the function code is not executed). If the function returns void, then the compiler will yield an error. We used *sizeof* to help us knowing the functions return type.

Retrieving such information is not trivial since *sizeof* does not handle functions returning *void* and the *typeid* class does not differentiate pointers from references. Besides the problem can not be solved in a dynamic way because of functions returning *void*. Indeed, we have to choose whether or not the return value is to be assign but the compiler will try to compute both cases and thus fail since they are antithetical.

We found a trick to handle void functions with *sizeof*. This trick was made possible thanks to the overloading of the C++ *comma operator*. Since *sizeof* is computed at compile time, it perfectly matches our needs.

The *operator,* is left-to-right associative and it can be overloaded in classes. When the overloading is invalid or does not exist, the compiler switches back to the C semantic of this operator: it simply evaluates every expression between the commas (from the left to the right) and the rightmost one gives its type to the whole expression.

```
func_returning_void(), func_returning_int(), func_returning_char();
```

We assume that those functions are respectively returning *void*, *int*, *and* char. The whole expression has the type *char* since the rightmost expression is a call to a function returning *char*. Now we introduce a class *foo* with an overloaded *operator,* and add a function returning *foo* in the comma sequence.

```
struct foo
{
  template <typename T>
  char operator, (T) {}
};
int main()
{
  func_returning_foo(), func_returning_int(); // type is char
  func_returning_foo(), func_returning_void(); // type is void
}
```

The first sequence is typed as *char*. Indeed, the first expression in the sequence is of type *foo* and *foo* has overloaded its *operator,* thus this operator is called with an *int* argument that comes from the second expression in the sequence. So the whole expression is typed as the return value of *foo::operator,* meaning *char*. The second sequence is typed as *void* because it is not valid to call *foo::operator,* with a *void* argument. Facing this, the compiler switches back to the C semantic and the whole expression has the same type as the rightmost sub-expression.

Using this property, it becomes possible to test whether or not a function returns void:

```
template <unsigned N>
```

```
struct helper
{
  char n[N];
  template <typename T>
  helper<N+1> operator,(T) { assert(0); }
};
helper<1> is_void;
void foo_void() {}
int foo_int() {}
struct whatever{};
int main()
{
  std::cout « sizeof(is_void, foo_void(), whatever()) « std::endl;
  std::cout « sizeof(is_void, foo_int(), whatever()) « std::endl;
}
```

Here is our trick, it consists of enclosing the function call between:

- One object the *operator,* of which is overloaded and returns an object with an overloaded *operator,*.
- An object the type of which does not matter since it is only used to give the expression its *size* when the function being tested returns void.

In this example, two cases are differentiated:

- the function returns void, then the comma-separated expression has the type of its rightmost operand. The size is then 1.
- the function does not return void, then the *operator,* of *is_void* is called on the value returned by *foo_int()* and returns a helper temporary object. The *operator,* of the temporary object is then called and returns a helper temporary object which size is 3.

The next step is to differentiate between pointers, constant pointers, references, constant references, copies, constant copies. This is a bit tricky since we can not overload the *operator,* for references in the same class that handles the values, it would produce a conflict. A new class is needed:

```
template <unsigned N>
struct helper
{
  char n[N];
  template <typename T>
  helper<N+1> operator,(T) { assert(0);}
};
template <unsigned N>
struct helper_ref
```

```
{
  char n[N];
  template <typename T>
  helper_ref<N+2> operator,(T) { assert(0);}
};
helper<1> is_void;
helper_ref<1> is_ref;
void foo_void() {}
int foo_int_ref() {}
int foo_int() {}
struct whatever_{} whatever;
int main()
{
  std::cout << sizeof(is_void, foo_void(), whatever) +
                     sizeof(is_ref, foo_void(), whatever)
                << std::endl; // displays 2
  std::cout << sizeof(is_void, foo_int(), whatever) +
                     sizeof(is_ref, foo_int(), whatever)
                << std::endl; // displays 4
  std::cout << sizeof(is_void, foo_int_ref(), whatever) +
                     sizeof(is_ref, foo_int_ref(), whatever)
                << std::endl; //displays 8
}
```

Now *whatever* needs to be a reference (at least in the *sizeof* containing *is_ref*) so that the *operator,* in *helper_ref* is called a second time. To handle functions returning pointers, one would just need to overload the *operator,* for pointers in the *helper_ref* struct. A complete implementation is available in our work (src/policy.hh).

This solution allows us to generate a wrapping function which will know the return type of the actual function to call. Thus the wrapping function will assign the return value if needed and will choose a satisfying allocation policy for the data proxy.

3.4 JIT module

The just in time module is split in three parts:

- the C++ generation unit
- the compilation unit
- the dynamic library loading, function pointer fetching, casting and calling

3.4.1 The C++ generation unit

It was done by a Ruby script for development performance reasons and was rewritten in C++ for time consumption performance reasons.

The generated code is quite short, it's after all just a function nested in an *extern "C"* block and in two namespaces *dyn::generated*. This function takes *const data* arguments and returns a data. The first job of this function is to extract the real value of each data argument using the knowlegde of its static type and a reinterpret_cast of the proxy. Then an assert is done on each casted pointer. Then the allocation policy is chosen according to the return type of the function (see Functions return type to better understand this). After that the call is done and the return value is saved according to the allocation policy. Finally the *data* is created and returned using the freshly built *data_proxy*.

The generated code is slightly different for methods which have the subject object as first argument and must be used with the good notation (o.m or o->m). For operators, the generated code is depending of the kind of the operator (prefix, infix or bracketed). For constructors we choose to generate a dynamic allocation policy using the *new* operator.

3.4.2 The compilation unit

The compilation is handled by a Ruby script and a *Makefile* generated by `automake`.

The script makes a directory by wrapper, it adds the source code in the file `function.cc` and copies the template Makefile to this directory.

The Makefile can build using `libtool` a good library suitable to dynamic loading via the `ltdl` library.

3.4.3 The dynamic library loading, function pointer fetching, casting and calling

The dynamic library loading is done using the `ltdl` library, and the function pointer fetching too. The pointer is casted and called according to the number of arguments. The function that handle this variable argument thing is just the *operator()* of the *fun* class that is overloaded N times according to a constant that makes code expansion using `erb`.

Chapter 4

Conclusion

4.0.4 Summary

Using C++ advanced techniques, we manage to develop a dynamic C++ environment to handily use static C++ classes and functions without facing the usual very long compilation times. We hope that it will make prototyping easier and that it will be as useful as we think it is.

4.0.5 Future work

The software is not completed yet, a few things still need to be done:

- We did not name the project, we sort of lacked imagination.
- The distribution needs to be improved.
- The code needs to be documented and cleaned up.

The package will have to be maintained because it is necessary to:

- cope with the future changes in the libiberty that we are using for demangling the C++ names.
- stay compliant with future versions of the `Gnu C++ Compiler`,
- take the users feedback into account.

4.1 Acknowledgments

We would like to thank `Thierry Géraud` for all the discussions we had with him about the design of Static/Dynamic Bridges and C++ tricks. Beside his feedback was greatly appreciated. We would like to thank `Roland Levillain` as well for all the time he spent helping us debugging our software.

Chapter 5

Bibliography

LRDE (1999). Olena, a generic image processing library written in c++.
olena.lrde.epita.fr.

LRDE (2002). The vaucanson project: a finite state machine manipulation platform written in c++.
vaucanson.lrde.epita.fr.

Thomas, D. (2004). *Programming Ruby: The Pragmatic Programmer's Guide, Second Edition*. Pragmatic Programmer.
pragmaticprogrammer.com/titles/ruby.

van Rossum, G. (1990). Python : interpreted, interactive, object-oriented programming language.
www.python.org.

Glossary

demangle:

see *compiler mangling*

functor:

In object oriented prgramming a functor is a object that responds to the operator() method. From the user side a functor behave like a function.

cache:

The cache is a memory place where you store computation results to avoid to recompute them.

lazy:

Unwilling to work or use energy. In the case of programming it's a technique that consist into compute values when they are really needed.

compiler mangling:

In the compiler domain the mangling is a transformation over objects like functions' prototype, variables' type, and so on into a compact symbol that can be used in an object file (.o). The demangling is the reverse operation that redraw the prototype from the symbol.

design pattern:

A design pattern is a "solution to a problem in context"; that is, it represents a high-quality solution to a recurring problem in design.

JIT:

Just in Time