Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Term Paper

Daniel Sarnow

Interface-based Programming in C++

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Sarnow

Title of the paper

Interface-based Programming in C++

Keywords

C++, Compiler Optimizations, Callgrind, Interface, polymorphism

Abstract

In many object oriented literature runtime polymorphism is often the only way mentioned to realize interface-based programming. In comparison to other languages, C++ offers more than just that. In C++ interface-based programming can also be achieved through link-time or compile-time polymorphism. This paper will show how interface-based programming can be done in C++ when using these three types of polymorphism. The different approaches are compared with each other at different compiler optimization levels. The paper will also show that compiler optimization mechanisms like inlining and devirtualization have been improved to minimize the overhead caused by dynamic dispatch even more.

Contents

List of Tables

List of Figures

Listings

1 Introduction

Like Stroustrup says polymorphism is the provision of a single interface to entities of different types[1]. In many object oriented literature runtime polymorphism (which means the use of inheritance and virtual functions) is often the **only** way mentioned to realize interface-based programming.

In comparison to other languages, C++ offers more than just that! In C++ interface-based programming can also be achieved through link-time or compile-time polymorphism.

The different types of polymorphism distinguish at which point in the process the implementation is selected.

This paper will show how interface-based programming can be done in C++, when using the three mentioned types of polymorphism.

Furthermore the different approaches will be compared with regard to their presumably runtime performance and memory usage at different optimization levels. Callgrind, a profiling tool that gathers information of the call history among methods, will be used to compare the different approaches.

[1] from http://www.stroustrup.com/glossary.html

2 Approaches

In comparison to other languages C++ offers a wide variety of possibilities to implement an interface. This chapter will give an overview on the different approaches that C++ offers.

The most common and probably the only way described in most object oriented literature is the use of inheritance and virtual functions to implement an interface. The decision which implementation to use will be made at runtime. This approach will be described in section 2.1 (Runtime Polymorphism).

The second possibility uses link-time polymorphism to determine the implementation that should be used. This approach will be described in section 2.2 (Link-Time Polymorphism).

The third approach concentrates on the use of compile-time polymorphism to make the decision on the implementation. This approach will be described in section 2.3 (Compile-Time Polymorphism).

The following table (2.1) shows when a certain interface implementation has to be used, because the information that is needed to decide on the implementation is only available at a certain point.

Decision **only** possible at …	Implementation to use …	Chapter
Run-Time	Dynamic Interface	2.1
Link-Time	Separately compiled sources	2.2
Compile-Time	Implicit Interface	2.3.1
Compile-Time	CRTP	2.3.2

Table 2.1: Necessity to use a certain interface implementation

2.1 Runtime Polymorphism

Probably the most common way to implement an interface in C++ is through the use of inheritance and virtual functions.

The class diagram 2.1 shows two derived classes that implement the interface / abstract base class *DynamicInterface*.

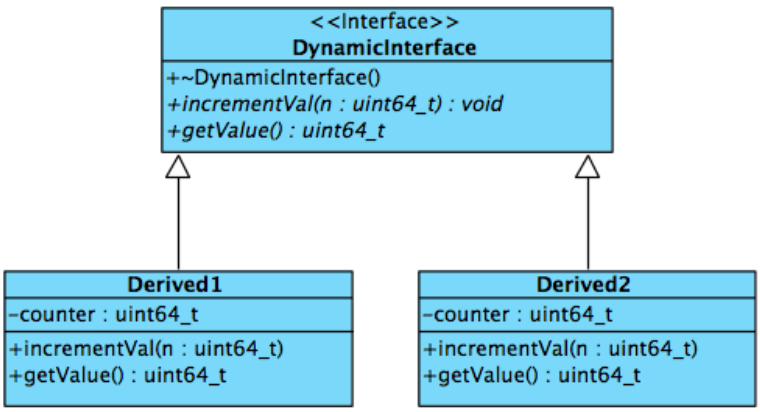

Figure 2.1: Class diagram: Dynamic Approach

The code for the abstract base class *DynamicInterface* is shown in listing 2.1. For a class to become an abstract class it has to have at least one pure virtual function. A pure virtual function is specified by the pseudo initializer '= 0' (for an example see line 4 and 5).

```
1 class DynamicInterface {
2 public:
3         virtual ~DynamicInterface(){};
4         virtual void incrementCounter(const uint64_t n) = 0;
5         virtual uint64_t getCounter() const = 0;
6 };
```

Listing 2.1: DynamicInterface.hpp

It is useful to have a virtual destructor in the base class, because that way it is possible to delete an instance of a derived class through a pointer to the base class. If the base class does not have a virtual destructor, the destructor of the base class might be called, which could lead to a resource leak.

Dynamic Dispatch

At runtime different implementations of the interface *DynamicInterface* can be used. Therefore dynamic dispatch or late binding has to be used to determine which implementation of a polymorphic method or functions has to be called.

A typically implementation of dynamic dispatch is implemented with the use of a data structure called a virtual table (*vtbl*).[1] A virtual table is a lookup table of pointers to functions and is used to resolve functions calls in dynamic dispatch.

Every class that uses virtual functions or is derived from a class that uses them has it's own virtual table. Every entry of this table is a pointer that points to the most derived function reachable by that class.

The compiler also adds a hidden pointer *vptr* to the base class that is set when an instance is created and points to the virtual table of the class. The vptr is also inherited by the derived classes and will point to the virtual table of the derived class.

The diagram 2.2 illustrated the just described mechanism.

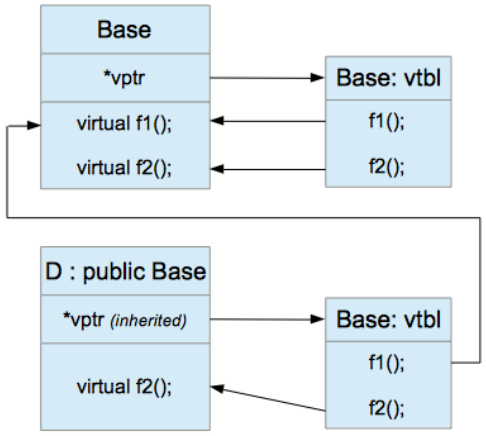

Figure 2.2: Dynamic Dispatch Mechanism

If there were an instance *d* of the derived class *D* and the function *f1* were to be called, the call would be handled by dereferencing *d*'s vptr, looking up the entry for *f1* in the virtual table and dereferencing that pointer to actually call the desired function.

2.2 Link-Time Polymorphism

Another possible approach for implementing an interface is through the use of link-time polymorphism.

[1] vtbls are not defined by the C++ standard. The implementation of dynamic dispatch can be compiler specific.

Like the name points out the selection of a certain implementation happens at the linking stage.

One possible scenario for this approach could be the code production for two different systems, one with expensive, high-performance components and one with cheaper, low performance components. And each component requires a different implementation.

This approach uses one class definition that serves both components (as an 'interface'). For each component an implementation of the 'interface' is written and compiled separately.

Depending on which implementation is needed or rather which component is used for the desired system, one of the separately compiled sources is linked with the main program.

2.3 Compile-Time Polymorphism

The last possibility to implement an interface is done by the use of compile-time polymorphism.

2.3.1 Implicit Interface

One way to achieve this is by using two or more classes that support the same implicit interface. An implicit interface is defined and each concrete implementation is it's own independent class. The 'interface' header file includes the header files from all implementations.

With the help of compile-time information, such as the size of a pointer (16, 32, 64 bit), a decision can be made on which implementation to use.

Computed typedefs can then be used to determine a certain implementation and set the type that is later used to instantiate the required object.

2.3.2 Curiously Recurring Template Pattern (CRTP)

An alternative possibility to implement an interface through the use of compile-time polymorphism is the Curiously Recurring Template Pattern (CRTP).

This pattern consists of a base class, the 'interface' (see listing 2.2), that is implemented by the derived classes like this:

class CRTPImplementation : public CRTPInterface<CRTPImplementation> { ... }

So far the process looks very similar to that of the dynamic implementation, the only difference is that no virtual functions are needed. The base class gets information on the real type of the derived class through a class template (line 1). That way the right function can be called without the need for virtual tables and dynamic dispatch.

```
1  template <typename Implementation>
2  class CRTPInterface {
3  public:
4    void incrementCounter(const uint64_t n) {
5      impl().incrementCounter(n);
6    }
7
8    uint64_t getCounter() {
9      return impl().getCounter();
10   }
11 private:
12   Implementation& impl() {
13     return *static_cast<Implementation*>(this);
14   }
15 };
```

Listing 2.2: CRTPInterface.hpp

At compile-time a cast to the type of the derived class can safely be done (line 13) and functions of the derived class can be called directly (no dynamic dispatch needed).

This approach achieves a similar effect as the use of inheritance and virtual function, without the cost of the dynamic dispatch, but also with the loss of some flexibility, because with CRTP the programmer has to decide already at compile-time which implementation to use.

3 Comparing the Approaches

From the chapter Approaches it is already known that a certain approach has to be implemented when the iselection of the implementation to use is only available at a certain point in the process (see table 2.1).

This chapter compares the different approaches with regard to their presumably runtime performance and memory usage at different compiler optimization levels.

In the following sections the dynamic approach (see section 2.1) and the Curiously Recurring Template Pattern (see section 2.3.2) will be compared to a 'normal' implementation (simple class with no virtual calls). The 'normal' implementation should behave similar to the approaches suggested in section 2.2 and 2.3.1.

3.1 Runtime Performance

In order to estimate the runtime cost of each approach a tool called callgrind will be used. Callgrind, a tool from the valgrind[1] tool collection, is a profiling tool that gathers information of the call history among methods and functions by generating a call graph.

A call graph is a directed graph that shows calling relationships between methods or functions of a program.

An example can bee seen in figure 3.1. The nodes represent methods or functions that are reachable within the program. The percentage in each box represents the time the program spends in that function.

The directed edges indicate the relationship between the functions. An edge is drawn from function *main* to function *run_normal*, because a call site in the function *main* calls the function *run_normal*. The numbers at each edge represent the number of times the function is called in the program flow.

[1] Valgrind is a tool collection for dynamically analyzing software. url:www.valgrind.org

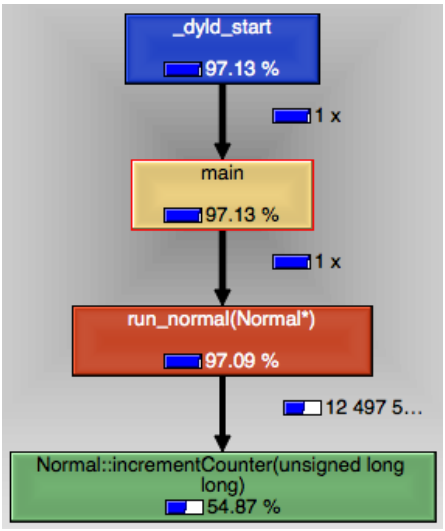

Figure 3.1: Call graph for the normal implementation at optimization level 0

In order to analyze the program valgrind lets the program run not directly on the host CPU, but in a virtual machine with a just-in-time compiler. That way code can be analyzed by the valgrind tools.

The following listing (3.1) gives an idea of the main function that was used for to each approach to collect the test data with callgrind.

```
1 const unsigned N = 5000;
2
3 void function(Approach* obj) {
4         for (unsigned i = 0; i < N; ++i) {
5                 for (unsigned j = 0; j < i; ++j) {
6                         obj->incrementCounter(j);
7                 }
8         }
9 }
```

Listing 3.1: Sample method for test runs

The diagram 3.2 is based on data from the callgrind analysis of each approach at different compiler optimization levels.

The y-axis represents the total number of instructions used by the program determined during the the callgrind analysis. In this specific case the more instructions a program has to execute the more time it will need to finish the task. Therefore a program that needs less instructions to solve the same problem as another program will be faster. The x-axis displays

the compiler optimization level. For this analysis GCC (GNU Compiler Collection), version 5.2.0 (released July 16, 2015) was used.

The first bar at each optimization level represents the dynamic approach using inheritance an virtual functions, the second bar the CRTP approach and the third bar the 'normal' implementation.

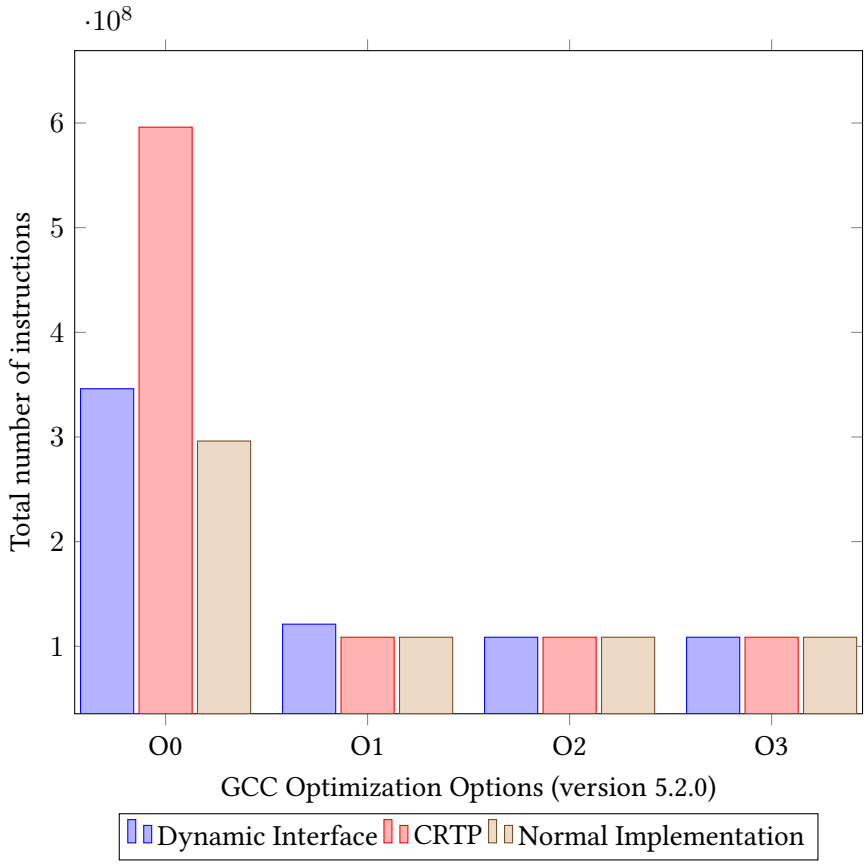

Figure 3.2: Affect of GCC Optimization Options (version 5.2.0) on the approaches

Overall the diagram (3.2) reveals that at a low optimization level there is a huge difference between the approaches and with rising optimization levels the differences between the approaches decrease to the point where they are almost gone.

The greatest difference can be seen at optimization level 0, which is the default optimization level in GCC [2]. At this stage the normal implementation is the fastest leaving the other ap-

[2] At level 0 not all optimization flag are disabled, a few optimizations are already in use. 'gcc -O*level* -Q –help= optimizers' shows which optimization flags are enabled at the chosen optimization level

proaches behind.

The dynamic approach has a slight overhead in comparison to the normal implementation, because as described in section 2.1 the correct function to call has to be identified by the dynamic dispatch mechanism, which is done at runtime an thus requires some computation time.

The CRTP approach has an even greater overhead. These additional instruction are not needed because of some dynamic dispatch code that has to determine the right function at runtime, because through the template this is already known at compile-time, but at level 0 there are a lot more functions to call until the actual function is executed.
The following lists shows the function calls that have to be made:

- CRTPInterface<CRTPImplementation>::incrementCounter(...)
  - CRTPInterface<CRTPImplementation>::impl()
  - CRTPImplementation::incrementCounter(...)

And this chain of function calls result in a far greater overhead leaving the CRTP approach behind.

At optimization level 1 and higher the normal and the CRTP approach seem to use about the same number of instructions and the dynamic approach still has some overhead, but it is also decreasing.
Important to notice is that all approaches have reduced their total number of instructions significantly.

Having a closer look, the callgrind analysis reveals that the actual function of the concrete implementation (*ConcreteImplementation::incrementCounter(...)*) is no longer called, only the function body is executed, which is *counter += n;*.
The reason for this behavior are mostly caused two sets of compiler optimization flags. One set of flags is known as inlining and the other is called devirtualization. The next subsection 3.1.1 will describe these mechanisms in more detail.

This example shows that if all important function calls are suitable for being optimized by the compiler that at optimization level 2 there is almost no more difference between the

different approaches.

To sum it up:
If the situation is not ideal, the approach that will probably perform the best is the normal implementation that uses a simple class with no virtual calls, because of the missing overhead right from the beginning.

In case of the dynamic approach there will probably be still some overhead for the dynamic dispatch, but as the the next section 3.1.1 will show, devirtualization has become quite successful and can eliminate quite a lot of virtual calls.

The CRTP Implementation suffered the most from the loss of optimization mechanisms and relies heavily on inlining by the compiler to perform properly and be competitive.

3.1.1 GCC Optimization Flags

This section will concentrate on the compiler optimization known as inlining and devirtualization.

Inlining

Inlining is a optimization mechanism that replaces the function call site with the body of the called function. This makes the execution usually faster, because the overhead on calling the function was eliminated. Besides the fact that inlining usually speeds up the program, it allows further optimization mechanism to be applied.

If any of the argument values of the function are constant, it may lead to a simplification of the function code at compile-time, so that only a part of the function's code need to be inserted at the call site and thereby an increase in the program's size caused by can be curtailed.

Functions that are not suitable to be inlined are those that make use of a variable number of arguments (also called variadic functions), the alloca() function, computed goto, nonlocal goto and the use of nested functions.

The programmer can use the keyword *inline* to tell the compiler that inlining is desired, but the compiler is not required by any means to follow that request.

With inlining usually also comes an increase in code size because of the duplication of the body of a function. So when it comes to the need of reduction of code size, like necessary in some embedded system's code, inline may not be used.

Inlining can also slow down the program instead of speeding it up. For example when large functions are inlined in a lot of places and the working set of the program does not fit anymore in one level (e.g. L1-cache) of the memory hierarchy.

Devirtualization

Devirtualization is a rather new mechanism and was first supported in the GCC version 4.7.0. When the flag devirtualize is enabled the compiler tries to convert virtual function calls to direct function calls, to do this the compiler has to determine the type of object which will call the function. The compiler will try to find all possible object types at the call site. If this set only contains one type or all types in the set use the same implementation the call site can be devirtualized.

Since GCC version 4.9.0 another flag called devirtualize-speculatively was introduced, which tries to convert virtual function calls to speculative direct function calls.

This mechanism will analyze the type inheritance graph and tries to identify for a certain function call a set of possible likely targets. If the set is small the function call can be changed to a conditional deciding between a direct and an indirect call. The good thing is that through the use of speculative calls more optimization mechanisms, like inlining, are possible.

In GCC version 5.2.0 devirtualization was improved even more and it was stated in the official change log that about 50% of the virtual calls in Firefox were speculatively devirtualized during link-time optimization.

3.1.2 Measurement of execution time of the Approaches

The last section concentrated on the evaluation of runtime cost via the total number of instructions that a program needs to solve a problem. It was claimed that in this case the greater the number of instructions a program needs the more computation time it requires.

To verify this and to get some real time measurements the same test was run on a Raspberry Pi 1, version B, revision 2. All unnecessary services were turned off and no desktop environment was used in order to minimize interferences by other processes. In order to get more

precise data each of the twelve time measurement tests were done 11 times and the median was used as a test result.

The table 3.1 presents the results of the execution on the Raspberry Pi.

Implementation	O0	O1	O2	O3
Dynamic Interface	1192	532	531	368
CRTP	1575	366	365	370
Normal	820	365	367	367

Table 3.1: Execution time in milliseconds

The data from table 3.1 and the data from the callgrind analysis (see figure 3.2) appear to be similar. This strengthens the claim that in this case their is a direct correlation between the total numbers of instructions and the programs speed.

When looking at the same test compiled with GCC version 4.6.3 (no devirtualization support) and run under the same circumstances the CRTP approach is at optimization level 2 about 3x faster than the dynamic approach. The reason is that the virtual calls could not be transformed to direct calls and therefore other optimizations like inlining could not be applied.

This shows that there has been a huge advancement in the compiler optimization features when it comes to the usage of virtual functions.

3.2 Memory Usage

Besides a programs runtime cost, another important indicator is a program's memory usage. With an increase in the program's performance through compiler optimization, for example through inlining, the memory size of a program generally also increases. For example when a larger function is called at different call sites and is inlined each time. This way the function body's code is duplicated many times. The analysis performed in section 3.1 (Runtime Performance) was a very simple test because the function used was very small, was only called at a single call site and therefore likely to be inlined.

In other scenarios were there could be more derived classes or concrete implementations the situation could be different.

The approaches that just use a simple class with no virtual calls (2.2 and 2.3.1) and the actual class is chosen at link-time or compile-time should not use more memory, because one class is simply replaced by the other.

In the case of the dynamic approach with inheritance and virtual functions (2.1) each class has an additional virtual pointer (vptr) that points to the virtual table, which also exists for every class (base and derived classes).

Figure 3.3 shows an extract from the class hierarchy dump for the dynamic implementation.

```
Vtable for DynamicImplementation
DynamicImplementation::_ZTV21DynamicImplementation: 6u entries
0     (int (*)(...))0
...
32    (int (*)(...))DynamicImplementation::incrementCounter
40    (int (*)(...))DynamicImplementation::getCounter

Class DynamicImplementation
   size=16 align=8
   base size=16 base align=8
DynamicImplementation (0x0x142d2c138) 0
    vptr=((& DynamicImplementation::_ZTV21DynamicImplementation) + 16u)
    ...
```

Figure 3.3: Extract from class hierarchy dump of dynamic implementation

In the CRTP approach each derived class implements the base class with a specific template that holds the information about the type of the derived class. But this also means that *CRTPInterface<DerivedA>* and *CRTPInterface<DerivedB>* are two different classes.

If a program's size is crucial, for example in code for embedded systems, CRTP is an approach that should probably be avoided and the simple class with computed typedefs approach (see section 2.3.1) should be preferred.

4 Conclusion

Sometimes the programmer is forced to use a certain approach, because the selection of a certain implementation can only be done at a certain point, for example at runtime when different instances of derived classes could be assigned to the same base class object's pointer depending on the program's progression. Because the selection can only be done at runtime the dynamic approach is required.

The table 4.1 indicates when the use of a certain approach is necessary.

Decision **only** possible at …	Implementation to use …	Remarks
Run-Time	Dynamic Interface (2.1)	A
Link-Time	Separately compiled Sources (2.2)	-
Compile-Time	Implicit Interface (2.3.1)	-
Compile-Time	CRTP (2.3.2)	B

Table 4.1: Necessity to use a certain interface implementation (II)

A The good news is that in recent time the overhead that comes with an implementation using virtual functions could be reduce significantly with the use of devirtualization. Depending on the complexity of the call site's code, it may be the case that not all virtual calls can be devirtualized.

B The CRTP may not be suitable in embedded environments, because of the possibility of increased code size. As seen in the analysis the CRTP relies heavily on the compiler optimization inlining. This could lead to an increase in code size, if larger functions are called at different call sites. Furthermore different implementation will also increase the code size, because every implementation will have it's own base class because the base class is a template class and the template is the type of the derived class.

Bibliography

[gccRelease] : *GCC 5 Release Series Changes*. – URL https://gcc.gnu.org/gcc-5/changes.html. – Zugriffsdatum: 2016-01-12

[gccInline] : *Inline (GCC)*. – URL https://gcc.gnu.org/onlinedocs/gcc/Inline.html. – Zugriffsdatum: 2016-01-8

[gccOptimize] : *Optimization Options (GCC)*. – URL https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc/Optimize-Options.html#Optimize-Options. – Zugriffsdatum: 2016-01-12

[stroustrup] : *Stroustrup: C++ Glossary*. – URL http://www.stroustrup.com/glossary.html#Gpolymorphism. – Zugriffsdatum: 2016-01-5

[valgrind] : *Valgrind Home*. – URL http://www.valgrind.org/. – Zugriffsdatum: 2016-01-12

[Bendersky 2011] Bendersky: *The Curiously Recurring Template Pattern in C++ - Eli Bendersky's website*. 2011. – URL http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/. – Zugriffsdatum: 2016-01-15

[Bendersky 2013] Bendersky: *The cost of dynamic vs. static dispatch in C++*. 2013. – URL http://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c. – Zugriffsdatum: 2016-01-15

[Driesen 1995] Driesen, Holzle: The Direct Cost of Virtual Function Calls in C++. In: *OOPSLA 96*, URL https://www.cs.ucsb.edu/~urs/oocsb/papers/oopsla96.pdf. – Zugriffsdatum: 2016-01-12, 1995

[Meyers 2014] Meyers, Scott: *Effective C++ in an Embedded Environment (Talk)*. 2014. – slides 76 to 93

[Namolaru 2006] Namolaru, Mircea: Devirtualization in GCC. In: *Proceedings of the GCC Developers Summit*, Citeseer, 2006, S. 125–133. – URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.307.1260&rep=rep1&type=pdf#page=131. – Zugriffsdatum: 2016-01-12