Selective Friends in C++

Gábor Márton*and Zoltán Porkoláb

Eötvös Loránd University, Dept. of Programming Languages and Compilers, H-1117 Pázmány Péter sétány 1/C, Budapest, Hungary.

SUMMARY

There is a strong prejudice against the friendship access control mechanism in C++. People claim that friendship breaks the encapsulation, reflects bad design and creates too strong coupling. However, friends appear even in the most carefully designed systems, and if it is used judiciously (like using the Attorney-Client idiom) they may be better choice than widening the public interface of the class.
In this paper we investigate how the friendship mechanism is used in C++ programs. We have made measurements on several open source projects to understand the current use of friends. Our results show various holes and errors in friend usage, like friend functions accessing only public members or not accessing members at all or the class which declare friends has no private members at all. The results also show that friend functions actually use only a low percentage of the private members they were granted to access, which is a source of errors.
These results have motivated us to propose a selective friend language construct for C++ which can restrict friendship only to well-defined members. Such a new language element may decrease the degradation of encapsulation and significantly increase the diagnostic capacity of the compiler. We have created a proof-of-concept implementation based on the LLVM/Clang compiler infrastructure to show that such constructs can be established with a minimal syntactical and compilation overhead. Copyright © 2011 John Wiley & Sons, Ltd.

1. INTRODUCTION

Encapsulation is one of the fundamental concepts in object-oriented programming [1]. It is used to distinguish between the specification (interface) and the implementation of a class, thus minimizing the dependencies among separately-written modules [2]. The interface is regularly defined in terms of possible services or methods the class offers to its clients. Methods are specified in forms of their signature. The implementation of a class defines the internal representation of its objects and the way its methods are implemented. The main idea behind encapsulation is to hide as many details of the representation of the classes as possible. This way we can greatly reduce coupling between objects of different classes, enforce a precise definition of class interfaces, and increase reusability. With the additional help of inheritance and polymorphism, we make possible the incremental development of software systems.

Strictly speaking, information hiding is not a mandatory part of encapsulation [3]. Encapsulation can be achieved by using different programming conventions and good programmers have already successfully applied these practices in various languages. However, language support for information hiding is essential in large-scale software projects, as the violation of encapsulation rules can be detected in an automated way.

---

*Correspondence to: Eötvös Loránd University, Dept. of Programming Languages and Compilers, H-1117 Pázmány Péter sétány 1/C, Budapest, Hungary. E-mail: martongabesz@gmail.com

In modern statically typed programming languages information hiding is usually implemented via specific access control rules. These *visibility rules* are typically compiler-checked static rules, therefore no or minimal runtime activity is involved. Most languages support module-based access control, but proposals have been made to refine the type system of these languages to implement object encapsulation too [4, 5, 6, 7, 8, 9]. In contrast, popular dynamically typed languages such as Python, Ruby, and Smalltalk provide very limited or no encapsulation at all. The implementation of object-oriented encapsulation for dynamically typed languages is an important research area [10].

In object-oriented languages the notion of the class is the main tool for both modularization and encapsulation. Accordingly, most of the access rules define the visibility of class members. Modern programming languages export services in an anonymous way; i.e. not naming the client classes able to use those services. An important exception is the Eiffel programming language, where *selective export* possibilities provide a fine-grained description to specify which class components are visible from individual client classes [11]. Unfortunately, most current popular object-oriented languages lack this feature. Java, C#, and C++ restrict us to enrol member access into visibility categories. In C++ we can use categories, like "exporting to all clients" (`public`), "exporting to the subclasses" (`protected`), and "exporting to none" (`private`). Though Java and C# add additional categories, they still rely on anonymous category-based export.

Schärli et. al. [3] pointed out that current object-oriented programming languages are surprisingly weak in terms of encapsulation mechanism provided for programmers. Their criticism emphasizes that access rights are inseparable from classes, and are not customizable. The authors also diagnose that access rights are specified in terms of fixed client categories, typically users and heirs (i.e. using public and protected categories).

In large projects, when the number of corresponding components is growing, the class interactions form exponential accretion. Here superfluous visibility is a persistent source of design, implementation, and maintenance problems. On the other hand, a fine-grained visibility strategy can utilize the full compiler possibilities of modern object-oriented languages to filter out unwanted access attempts.

The C++ programming language has a special feature called `friend` declaration to specify individual methods or full classes to access non-public members in a class. Unfortunately a friend has permission to access every field, method, nested class, etc. with private or protected visibility. This behaviour essentially switches off all possible automated detection of access violations, i.e. when a friend accesses wrong members.

1.1. C++ Friends

In C++, *friend* is a special language element for access control mechanism [12, section 11.3]:

> A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations.

Example:

```cpp
class A {
  int x = 0;
  friend void foo(A &a);
};

// Accessing private member:
void foo(A &a) { a.x = 42; }
```

A *befriending class* is a struct/class which declares one or more friend function, friend function template, friend class or friend class template. Friend declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class.

Friendship is a great tool to enhance encapsulation in some special cases [13]. We often need to split a class in half when the two halves will have different numbers of instances or different lifetimes. In these cases, the two halves usually need direct access to each other. The safest way to implement this is to make the two halves friends of each other. Friends have been used also

for providing better syntactics [14]. Consider for instance the binary infix arithmetic operators: `aComplex + aComplex` should be defined logically as part of the `Complex` class. However, `operator+` has to be a free standing function to support `aFloat + aComplex` order of arguments as well, thus we declare it as a friend. Another examples for better syntax are the stream operators (`<<`, `>>`) which often have to access the internals of a class and they have to be free functions.

Friend is an explicit mechanism for granting access, just like membership. Member functions and friend functions are equally privileged, they access every innards of a class. The major difference is that a friend function is called like `f(x)`, while a member function is called like `x.f()` [13, 15]. Stroustrup states that during the language design, a friendship declaration was seen as a mechanism similar to that of one protection domain granting a read-write capability to another. It is an explicit and specific part of a class declaration [16, 2.10].

When we design a class, we try to minimize the number of functions that access the representation of the class and try to make the set of access functions as appropriate as possible. Therefore, the first question regarding to a (candidate member) function is whether it really needs access to the class? Typically, the set of functions that need access is smaller than we are willing to believe at first [15, 19.4.2]. Meyers states that given a choice between a member function (which can access not only the private data of a class, but also private functions, enums, typedefs, etc.) and a non-member non-friend function (which can access none of these things) providing the same functionality, the choice yielding greater encapsulation is the non-member non-friend function, because it does not increase the number of functions that can access the private parts of the class [17, Item 23]. Sutter and Alexandrescu have similar statements [18, chapter 44]. In other words, if we have a friend function which does not access any private entities it weakens the encapsulation since it increases the number of functions that can access the internals. By declaring that function non-friend, the encapsulation is not weakened and the functionality can be preserved since there is no need to access private entities. Consequently, we refer a friend function (or a method of a friend class) as *erroneous* or *superfluous* if it is not used to provide better syntax — like the non-member `operator+` — and it does not access any private or protected member.

Because friends can be easily misused [16, 19] some people discourage the use of friends at all [20, 21]. The different use cases of friendship and the criticism about it have motivated us to investigate friend usage in real world, open source projects.

This article is organized as follows: in section 2 we overview other languages' friendship like capabilities. In section 3 we describe the measurements we made and we summarize the results. We present and analyze our new friendship proposal (selective friend) in section 4. Earlier measurements on friend usage and alternatives of C++ selective friend is discussed in section 5. We describe possible future research in section 6. Our paper concludes in section 7.

2. FRIENDSHIP IN OTHER PROGRAMMING LANGUAGES

2.1. Java

In the Java language, there is no such thing as friendship. However members without any access modifier (package-private) are accessible via all those classes which are declared in the same package [22]. This mechanism cannot control access in such a fine-grained way as friendship, but for most of the cases it is good enough. For example, a white-box test [23, 24] might want to have access to the class it is testing. This can be achieved by declaring the members of the class package-private and by adding the test to the same package.

For a more sophisticated, friendship like access control we can use the workaround presented at Figure 1 [25]. Here, class `Owner` has a privileged function which shall be accessed only by the `User` class. The nested and public `User.Key` class has a private constructor which is accessible only in the parent class (`User`), therefore the `User.Key` object can be constructed only by the `User` class. The `User.usePrivileged()` function uses a static instance of the mentioned Key. The `Owner.privileged()` function requires an instance of the `User.Key`

```
import java.util.*;
import java.lang.*;
import java.io.*;

class Owner {
  public static void privileged(User.Key l) {
    l.hashCode();
    System.out.println("Privileged function called.");
  }
}

class User {
  public static class Key {
    private Key() {}
  }
  private static Key key = new Key();
  public static void usePrivileged() {
    Owner.privileged(key);
  }
}

class Main {
  public static void main(String[] args) throws java.lang.Exception {
    User.usePrivileged();
    // Owner.privileged(null); // runtime error
    // Owner.privileged(new User.Key());
    //                  ^^^^^^^^^^^compile error
  }
}
```

Figure 1. Friendship like access control in Java

class to be passed. Putting this together, only the methods of the User class will be able to call Owner.privileged(). If we call Owner.privileged() with a null pointer, then we will receive a runtime error. If we try to create an instance of User.Key not in the scope of User class, then we will get a compile error.

This technique cannot be implemented either in C++, or in C#, because it relies on a special access rule, which is unique for Java. Namely, an enclosing class can access its nested classes' private members [26, 27]. With this approach, we can precisely control which classes can access privileged functions of a class. By evolving further this technique, we can also control which specific member functions are accessible and this is more fine-grained access control than C++ friendship is. We can restrict access of friends only to a specific set of member functions in C++ as well, with a similar idea which also uses private constructors and special Key classes. Albeit, those Key classes need to declare some auxiliary friends. Later in section 5.3 we elaborate this approach. This methodology is capable of handling only member function access, but it cannot handle member field access.

2.2. CSharp

In C# there is no friend keyword as well, though we can define members as internal, thus making them available to the current assembly [28]. This is very similar to what Java has with package-private access level.

Also there is the InternalsVisibleToAttribute with which we can achieve a behaviour that is very similar to friendship [29]. With this attribute we are able to declare other assemblies to be friends of our assembly. The access control is happening on assembly level and we cannot be more specific to control the access class by class [30].

Figure 2 is an example that uses the InternalsVisibleToAttribute attribute to make an internal member of an unsigned assembly visible to another unsigned assembly. The attribute ensures that the internal StringLib.IsFirstLetterUpperCase method in an assembly named UtilityLib is visible to the code in an assembly named Friend2. We can see an example in Figure 3 which provides the source code for the Friend2 assembly. Note that if we are compiling in C# from the command line, we must use the /out compiler switch to ensure that the name of the friend assembly is available when the compiler binds to external references.

```
using System;
using System.Runtime.CompilerServices;

[assembly: InternalsVisibleToAttribute("Friend2")]

namespace Utilities.StringUtilities
{
   public class StringLib
   {
      internal static bool
      IsFirstLetterUpperCase(String s)
      {
         string first = s.Substring(0, 1);
         return first == first.ToUpper();
      }
   }
}
```

Figure 2. Use of InternalsVisibleToAttribute in CSharp (UtilityLib.dll)

```
using System;
using Utilities.StringUtilities;

public class Example
{
   public static void Main()
   {
      String s = "The␣Sign␣of␣the␣Four";
      Console.WriteLine(StringLib.IsFirstLetterUpperCase(s));
   }
}
```

Figure 3. Use of InternalsVisibleToAttribute in CSharp (Friend2 assembly)

```
module X;
class A {
  private:
    static int a;

  public:
    int foo(B j) { return j.b; }
}
class B {
  private:
    static int b;

  public:
    int bar(A j) { return j.a; }
}
int abc(A p) { return p.a; }
```

Figure 4. Accessing private members in D (within the same module)

2.3. Other Languages

D is a systems programming language with C-like syntax and static typing. It combines efficiency, control and modeling power with safety and programmer productivity. In this language, friend access is implicit in being a member of the same module [31]. This means, a member which is declared private can be accessed by other classes of the same module (Figure 4). The private attribute prevents only other modules from accessing the members.

Rust is another systems programming language [32, 33]. In addition to public and private, Rust allows users to declare an item as visible within a given scope (Figure 5). This mechanism is similar to friendship, but we grant access to whole modules not to just simple functions or classes. The rules are as follows:

```
pub mod outer_mod {
    pub mod inner_mod {
        // This function is visible within 'outer_mod'
        pub(in outer_mod) fn outer_mod_visible_fn() {}

        // This function is visible to the entire crate
        pub(crate) fn crate_visible_fn() {}

        // This function is visible within 'outer_mod'
        pub(super) fn super_mod_visible_fn() {
            // This function is visible since we're in the same 'mod'
            inner_mod_visible_fn();
        }

        // This function is visible
        pub(self) fn inner_mod_visible_fn() {}
    }
    pub fn foo() {
        inner_mod::outer_mod_visible_fn();
        inner_mod::crate_visible_fn();
        inner_mod::super_mod_visible_fn();

        // This function is no longer visible since we're outside of 'inner_mod'
        // Error! 'inner_mod_visible_fn' is private
        //inner_mod::inner_mod_visible_fn();
    }
}
```

Figure 5. Rust: declaring items visible within a given scope

- `pub(in path)` makes an item visible within the provided path. `path` must be a parent module of the item whose visibility is being declared.
- `pub(crate)` makes an item visible within the current crate. (A crate is a unit of compilation and linking, as well as versioning, distribution and runtime loading.)
- `pub(super)` makes an item visible to the parent module.
- `pub(self)` makes an item visible to the current module.

Script languages like Python and Perl do not have any access restriction mechanism [34, 35]. However there is a convention to name class internal variables with an underscore prefix. The lack of access restriction sometimes results unbeneficial workarounds. For instance, in case of one implementation of the Singleton pattern an exception is raised if the Singleton object is already instantiated (since there is no way to make the constructor private) [36].

3. MEASUREMENT

For better understanding of the current use of friends in C++ we decided to carry out empirical research. We were interested in (i) how many private members are accessed by friend functions, (ii) what is the ratio of accessed per non-accessed private members, (iii) are there indications of erroneous friend usage. To execute the measure we implemented a utility based on the LLVM/Clang compiler infrastructure' LibTooling and LibASTMatchers libraries [37]. With the help of these libraries we can derive statistics from the abstract syntax tree (AST) of the examined source files. Our utility is publicly available [38].

3.1. Description of the Measurement Algorithm

We can collect the statistics just for one translation unit (TU), or for a whole project with several translation units. If there are more than one TUs in the measure, we pay attention to not count twice those friend declarations whose definitions are in headers.

During the measurement we avoid collecting information on unused (i.e. neither instantiated, nor explicitly specialized) templates in all contexts. If the befriending class is a class template, then it

```
1   class A {
2     template <typename T>
3     friend void func(T, A& a);
4   };
5   template <typename T>
6   void func(T, A& a) {}
7
8   // Full(explicit) specialization
9   template <> void func<double>(double, A& a) {}
10
11  // Explicit instantiations
12  template void func<int>(int, A&);
13  template void func<double>(double, A&);
14
15  // Implicit instantiation
16  void foo() { A a; func<double>(1.0, a); }
17
18  // An other primary template
19  template <typename T> void func(T*, A&) {}
```

Figure 6. A more complex example for measurement

must have at least one instantiation/specialization to participate in the measure. We do not measure partial specializations.

We generate the statistics from measurement entries, which are stored for each friend entity. We store a *function measurement entry* for each friend function definition. If that friend declaration is a function template, then we create an entry for each different template specialization/instantiation. It means we do not examine the primary template itself, but only the specializations/instantiations. So, not used (primary) function templates will not affect the collected statistics.

In case of friend classes, we store a *class measurement entry*. If we find a friend class template, then we examine only its specializations/instantiations. For every friend class or class template specialization/instantiation we examine all of its member functions and member function templates and we create a function measurement entry for each of them. The member function templates are handled similarly as we handle the simple friend function templates, i.e. only the specializations/instantiations are checked, not the primary function template.

If the friend class or class template specialization/instantiation has nested classes, then we collect statistics from all of the functions and function templates of the nested classes. The rules regarding to the function instantiation and specialization are the same as in case of non-nested classes. If the nested class happens to be a class template, then we check only the specializations/instantiations of it. Again, in cases of class templates, we do not check partial specializations. Only explicit specializations and explicit/implicit instantiations are counted.

For example consider the below code:

```
class A {
  friend class B;
};
class B {
  void func(A &a) {}
  template <typename T>
  class C {
    void func(A &a) {}
  };
};
template class B::C<int>;
```

Here, the measurement will contain two class measurement entries; one for `class B` and one for `class B::C<int>` instantiation. Each class measurement entry contains one function measurement entry.

A more complex example follows in Figure 6. Examining this translation unit, we will have two function measurement entries. One for the specialization/instantiation with `double` and one for the instantiation with `int`. The instantiations in line 13 and 16 are superfluous instantiations, since the compiler has already created the corresponding representation for those instantiations via the

specialization in line 9. The last line of the example (line 19) defines a new primary template which is independent from the friend function template.

Note, there is no such thing as partial function template specialization in C++, therefore

```
template <typename T> void func(T, A& a) {}
template <typename T> void func<T*>(T*, A&);
```

would be illegal. What is more, it is advised to not specialize function templates at all [18, chapter 66].

Each function measurement entry contains the number of used private or protected

- member variables (including static variables),
- member functions (including static functions),
- types

in that particular function or function template specialization/instantiation. We refer private or protected member variables, member functions and nested types as *private entities*.

Also, each entry contains the number of private or protected entities in the corresponding befriending class.

For instance:

```
class A {
  int a = 0;
  int b;
  int c;

  friend bool operator==(A x, A y)
  {
    return x.a == y.a;
  }
};
```

Here, we have one function measurement entry for `operator==`; the number of used private member variables is 1 (variable `a`); the number of private member variables in the befriending class is 3 (variables `a`, `b`, `c`).

All the statistics are derived from function measurement entries. Statistical values are calculated separately for friend classes and for friend functions.

Under the term *friend function instances* we refer:

- friend functions
- friend function template specializations/instantiations

of the befriending class. We name those friend function instances which use at least 1 private entity as *correct friend function instances*. Under the term *friend class function instances* we refer:

- friend classes' member functions
- friend classes' nested classes' member functions
- the specializations/instantiations of the above two, if they happen to be function templates
- member functions of friend class template specializations/instantiations
- the specializations/instantiations of the above, if it happens to be a function template
- member functions of nested class template specializations/instantiations of friend class template specializations/instantiations
- the specializations/instantiations of the above, if it happens to be a function template

of the befriending class.

Under the term *friendly function instances* we refer all the friend function instances plus all the friend class function instances of the befriending class. We name those friendly function instances which use at least 1 private entity as *correct friendly function instances*. The *private usage* of a friendly function instance is the number of the used private entities in that particular function. The *private usage ratio* of a correct friendly function instance is the number of the used private entities divided by the number of all the private entities of the befriending class. We interpret this number

only on correct friendly functions, so it is always greater than 0.0 and less than or equal to 1.0 (as a correct friendly function uses at least 1 private entity, which implies that its befriending class has at least 1 private entity). The private usage of the `operator==` in the previous example is 1; the private usage ratio is $\frac{1}{3}$.

3.1.1. Avoiding duplicated measurement entries We must not count friend declarations multiple times if they are defined in a header and there are multiple source files including this header. Consider the following 3 files:

```
// header file: a.h
class A { friend void func(); }; void func(){};

// source file: TU_A.cpp
#include "a.h"

// source file: TU_B.cpp
#include "a.h"
```

In this case we provide only one function measurement entry. That entry is reachable via an associative container and identified by the source location of the friend declaration (`a.h:1:23`).

We need to provide extra care in order to avoid the counting of the same instantiations or specializations multiple times. We need to be attentive both when the befriending class is a template and when the friend function itself is a template. For example consider the following header file:

```
1   // header file: a.h
2
3   template <typename T> class A;
4
5   template <typename T> void func(A<T> &a);
6
7   template <typename T> class A {
8     int a = 0;
9     int b;
10    int c;
11
12    // refers to a full specialization
13    // for this particular T
14    friend void func<T>(A &a);
15  };
16
17  template <typename T>
18  void func(A<T>& a) {
19    a.a = 1;
20  }
```

We forward declare the class template `A` (line 3), then we forward declare the function template `func` that takes a reference to the object of one instantiation of the class template `A` (line 5). Later, we define the class template (line 7-15). Inside the class definition we declare a friend function for each instantiations of `A` (line 14). The definition of the function template follows (line 17-20).

Then, let us imagine we have the following two source files, each of them is compiled into a separate translation unit:

```
// source file: TU_A.cpp
#include "a.h"
template void func(A<int>& a);

// source file: TU_B.cpp
#include "a.h"
template void func(A<int>& a);
```

Both of them contain the very same implicit instantiation of the class template, but our measurement shall not count twice. Also, the function template is explicitly instantiated twice, but we shall avoid adding the very same function measurement entry twice to the friend declaration. To achieve this, we need to identify the instantiations in a unique way. We use the diagnostic name of the identifiers, which is the human readable form of the mangled names. We get this by calling the `getNameForDiagnostic()` function on the different `Decl` classes provided by Clang. The above set of the three files compiled into a project produces the following function measurement entry:

```
befriending class: A<int>
friendly function: func<int>
friendDeclLoc: a.h:14:15
defLoc: a.h:18:6
diagName: func<int>
usedPrivateVarsCount: 1
parentPrivateVarsCount: 3
usedPrivateMethodsCount: 0
parentPrivateMethodsCount: 0
types.usedPrivateCount: 0
types.parentPrivateCount: 0
```

So we have one function measurement entry registered for the one friend declaration (which is identified by its source location).

However, each friend function template could have different specializations with their own definition. Therefore, for one friend declaration we should register more than one measurement entries. So we store measurement entries for such different specializations in an associative container, with a tuple of the name of the befriending class and the actual specialization as a key. (We need to be able to search this container to avoid duplicates.) For instance if the header file is the same as before but the two source files are the following, than we will have two function measurement entries:

```cpp
// source file: TU_A.cpp
#include "a.h"
template void func(A<int>& a);

// source file: TU_B.cpp
#include "a.h"
template <class T>
struct Z { using type = char; };
template <class T>
using XYZ = Z<T>;
template void func(A<XYZ<char>::type>& a);
```

We can see that the canonical type of XYZ<char>::type is actually simple char. Therefore, we get one entry for func<char> and one for func<int>:

```
befriending class: A<char>
friendly function: func<char>
friendDeclLoc: a.h:14:15
defLoc: a.h:18:6
diagName: func<char>
// ...

befriending class: A<int>
friendly function: func<int>
friendDeclLoc: a.h:14:15
defLoc: a.h:18:6
diagName: func<int>
// ...
```

The keys with which we store these entries are the pairs (A<char>, func<char>) and (A<int>, func<int>).

The layout of the measurement structure for friend functions has the following form:

```cpp
using FriendDeclId = std::string;
using BefriendingClassInstantiationId = std::string;
using FunctionTemplateInstantiationId = std::string;
using FuncResultKey = std::pair<BefriendingClassInstantiationId,
                                FunctionTemplateInstantiationId>;
using FuncResultsForFriendDecl = std::map<FuncResultKey, FuncResult>;

std::map<FriendDeclId, FuncResultsForFriendDecl> FuncResults;
```

The above consequences are similar in case of friend classes, therefore we just present the layout of the measurement structure for the friend classes in Figure 7.

3.1.2. Special purpose friend functions Some degenerated use of friendship has evolved over the years. One such usage of friend functions is to define free functions inside the declaration context of the befriending class, i.e. to define free functions in-class. In this case access control is not

```
struct ClassResult {
  std::string diagName;
  FuncResultsForFriendDecl memberFuncResults;
};
using ClassTemplateInstantiationId = std::string;
using ClassResultKey = std::pair<BefriendingClassInstantiationId,
                                 ClassTemplateInstantiationId>;
using ClassResultsForFriendDecl =
    std::map<ClassResultKey, ClassResult>;

std::map<FriendDeclId, ClassResultsForFriendDecl> ClassResults;
```

Figure 7. Measurement structure for friend classes

considered at all, only the secondary property of a friend function definition is used. For example the Boost.Operators [39] library uses friend functions and CRTP [40] to generate free functions for client classes which derive from the library classes.

In the following listing, a greater-than operator is automatically added, even though there is no declaration, because the greater-than operator can be implemented using the already defined less-than operator [41]:

```
struct animal : public boost::less_than_comparable<animal> {
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
  bool operator<(const animal &a) const { return legs < a.legs; }
};

int main() {
  animal a1{"cat", 4};
  animal a2{"spider", 8};
  std::cout << std::boolalpha << (a2 > a1) << '\n';
}
```

The corresponding generator class has the following form:

```
template <class T, class B = ::boost::detail::empty_base<T>>
struct less_than_comparable1 : B {
  friend bool operator>(const T &x, const T &y) { return y < x; }
  // ...
};
```

We might wonder why the library does not provide a free function template instead of an in-class defined friend function, like below:

```
template <class T>
bool operator>(const T &x, const T &y) {
  return y < x;
}
```

If it did that, then there would be a possible instantiation for all types, not just for the type we really want to have the greater-than operator generated.

Another special purpose use of friends is to overcome some language constraints during template argument deduction. For example, let us assume we want to support implicit conversions on all operands of an operator of a class template [17]:

```
template <typename T> class Rational {
public:
  Rational(const T &numerator = 0, const T &denominator = 1);

  const T numerator() const;
  const T denominator() const;
  // ...
};

template <typename T>
const Rational<T> operator*(const Rational<T> &lhs,
                            const Rational<T> &rhs) {
  // ...
}
```

We want the code below to compile:

```
Rational<int> oneHalf(1, 2);
Rational<int> result = oneHalf * 2; // error!
```

However, the second line will not compile. The reason behind that is the compiler tries to deduce the `T` template type parameter of `operator*`, but it cannot. The deduction succeeds for the first parameter (`oneHalf`), but it fails for the second parameter. We might expect the compiler to use `Rational<int>`'s non-explicit constructor to convert 2 into a `Rational<int>`. But it does not do that, because implicit type conversion functions are never considered during template argument deduction.

The solution for this problem is to define the `operator*` as a friend function inside the body of the `Rational` class template:

```
template <typename T> class Rational {
public:
  // ...

  friend const Rational operator*(const Rational &lhs,
                                  const Rational &rhs) {
    // ...
  }
};
```

Now the call to `operator*` will compile, because when the object `oneHalf` is declared to be of type `Rational<int>`, the template class `Rational<int>` is instantiated, and as part of that process, the friend function `operator*` that takes `Rational<int>` parameters is automatically declared. As a declared function (not a function template), compilers can use implicit conversion functions (such as `Rational`'s non-explicit constructor) when calling it, and that is how they make the call succeed. As Meyers writes in [17]:

> The use of friendship has nothing to do with a need to access non-public parts of the class. In order to make type conversions possible on all arguments, we need a non-member function; and in order to have the proper function automatically instantiated, we need to declare the function inside the class. The only way to declare a non-member function inside a class is to make it a friend.

This use relies on the side effects of friend declarations, and it falls far away from the original intention of friendship, which was to provide sophisticated access control.

We refer a friend function as *Meyers candidate* if it has the following properties:

- It has zero private usage.
- Its befriending class is a class template.
- The function is defined inside (in-class) the befriending class.

The friend functions in both of the above examples (`operator>` in `less_than_comparable1` and `operator*` in `Rational`) fulfills these properties. Meyers candidates are very likely to be using the `friend` declaration specifier to define in-class free functions for a specific purpose and not for accessing private entities.

3.2. Measurement Results

We have measured four open source projects:

1. Boost Libraries [42], version 1.56.0, ~2.0 million lines of C/C++ code
2. LLVM and Clang [43], version 3eec7e6 (llvm.org/git/clang.git), ~2.3 million lines of C/C++ code
3. ITK [44], version 4d37786 (itk.org/ITK.git), ~1.0 million lines of C/C++ code
4. QT (qtbase package, core module) [45], version 5.6.3, ~200 thousand lines of C/C++ code

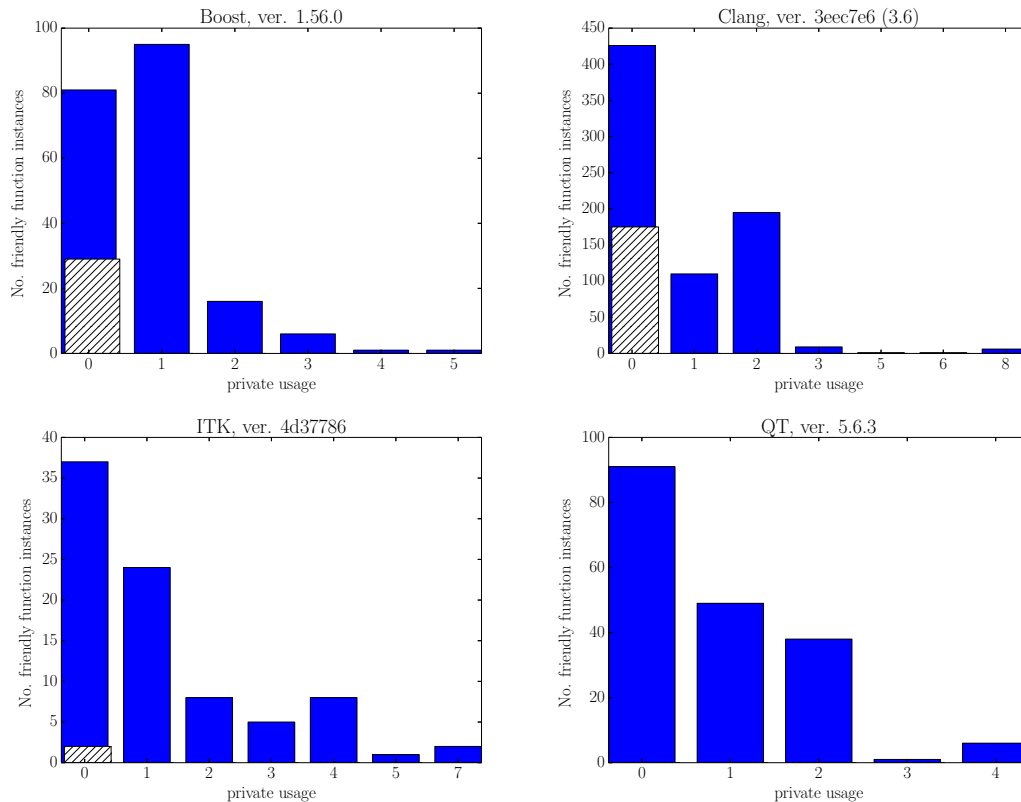The detailed measurement logs can be accessed publicly [38].

Figure 8. Private usage in functions

Figure 8 shows the average private usage in friend function instances for these projects. We can see for example in case of the Boost libraries there are around 95 friend functions which use one private entity. With the shaded (white) bar we display the number of the Meyers candidates. We do not draw bars for those results where the number of friends is zero; for instance there is no bar for 7 in case of the Clang project. We can see that there are lots of friend function instances which do not use any of the private entities (zero private entity usage). There can be two reasons behind this:

1. The befriending class does not have any private entities.
2. The friend function instance simply does not use any of the private entities. Note, those friend function instances which are in this category, might be in the previous category as well, because a friend function cannot use any of the private entities if the befriending class does not have any.

A surprisingly big portion of the friend function instances are not using any of the private or protected members. They are declared as friends either by mistake or during maintenance their implementation has changed to not access any privates. Or they have been declared as friends for some special reason which is not connected to access control, i.e. they are Meyers candidates.

According to Stroustrup we should strive to minimize the number of functions that access internals [15, 19.4.2]. Meyers states the encapsulation is greater if there are fewer functions that can access the private parts of the class [17, Item 23]. Consequently, we consider all the those non-Meyers-candidate friend functions as erroneous friends which have zero private usage.

Those friend functions who are Meyers candidates shall not be counted as erroneous friends. Even though they do not access any private entities they might had been declared as friends to define a free function. However we cannot know if this is the only true case. It might as well be possible, that the function had accessed privates previously, but during maintenance it has been changed to

```
struct uchar_wrapper {
  //... // there is no any access specifier

  // Friend function definition
  friend std::ptrdiff_t
  operator-(uchar_wrapper const &lhs, uchar_wrapper const &rhs) {
    return lhs.base_cursor - rhs.base_cursor;
  }

  //...
};
```

Figure 9. An erroneous friend declaration in Boost

not access privates anymore. Nevertheless, the number of the Meyers candidates gives us the upper limit of the error of our measurement regarding to erroneous friend functions.

We conclude that even considering the Meyers candidates there can be many friend functions or classes which are declared as friends superfluously, this way weakening the encapsulation. Therefore, we created a tool [38] with which we can list all the possibly erroneous friend declarations in a project. More specifically this tool can list

1. All those befriending classes which have at least one friend declaration but itself does not have any private entities. Note we do not list a class if all of its friend declarations are functions and all of those functions are Meyers candidates.
2. All those friend classes whose all member functions do not access any private entities in the befriending class (and the befriending class has private entities).
3. All the friend functions which might not be friend because they do not access any private entities. We list only those functions whose befriending class have private entities and it is not a Meyers candidate.

(Note, this tool is also based on Clang LibTooling library and shares a lot of common source code with the statistical measurement tool). With the help of the tool we could identify those friends in the Boost, Clang, ITK and QT projects that were declared as friend by mistake. For instance, in the ITK library there is a befriending class, which declared the operator<< as a friend function:

```
class GDCM_EXPORT Sorter {
  friend std::ostream &
  operator<<(std::ostream &_os, const Sorter &s);

public:
  // ...
  void Print(std::ostream &os) const;

protected:
  std::vector<std::string> Filenames;
  // ...
};

inline std::ostream &operator<<(std::ostream & os, const Sorter &s) {
  s.Print(os);
  return os;
}
```

Still, the definition of operator<< uses only the public Print() function and does not use any of the private entities.

We can find an other example for erroneous friend usage in the Boost libraries as well, listed in Figure 9. The uchar_wrapper struct does not have any explicit access specifier in the body of the class definition, i.e. all the members are public. Still, the operator-() is declared as friend. These cases safely can be considered as misuses of the friend construction. Note, currently there is no language construction in C++ to enable the compilers to detect and warn about such situations.

Figure 10 show the average private usage in friend class function instances for Boost, Clang, ITK and in QT corelib. The number of friend class function instances with zero private entity usage is quite high. In all four libraries, the number of these class function instances is significantly larger than the number of the correct class function instances. There is a huge number of member functions
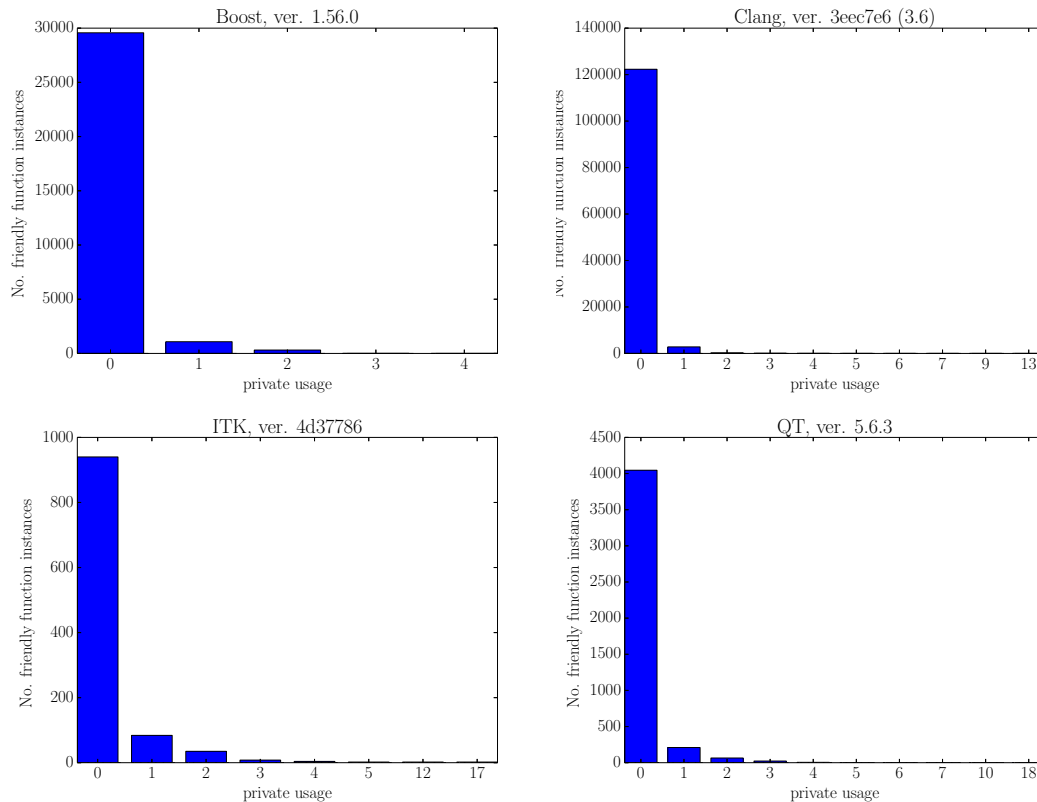
Figure 10. Private usage in classes

in friend classes which does not use any private members but they have access for all of the private entities of the befriending class. Compared to friend functions, the ratio of non-correct / correct function instances is way higher in case of friend classes. This means that friend classes provide access for a wide range of functions that are potentially unrelated to the befriending class. Therefore we consider the use of friend classes more harmful to encapsulation than the use of friend functions.

Figure 11 presents the average private usage ratio of correct friend function instances while figure 12 displays the average private usage ratio of correct friend class function instances. We can see, there are lots of the correct friendly function instances which are just accessing only a small portion of the befriending class' private entities. Also, in Figures 8 and 10 we can see that the vast majority of the correct friendly function instances access only 1-4 private entities. I.e. those functions which do access internals usually do not access more than a few members. In other words the library authors allowed unnecessary access to a large number of private entities.

4. SELECTIVE FRIEND

4.1. A New Lingual Element

Meyers states the encapsulation is greater if the number of functions that can access the private parts of the class is fewer [17, Item 23]. Similarly, we state that the encapsulation is greater if the number of accessible private entities is fewer because in that case the accessible private part of the class is smaller.

We have seen in section 3.2 that in Boost, Clang, ITK and QT the vast majority of correct friendly function instances access only 1-4 private entities although the number of private entities in the befriending class is usually much higher. Hence, we conclude there are many friend functions who
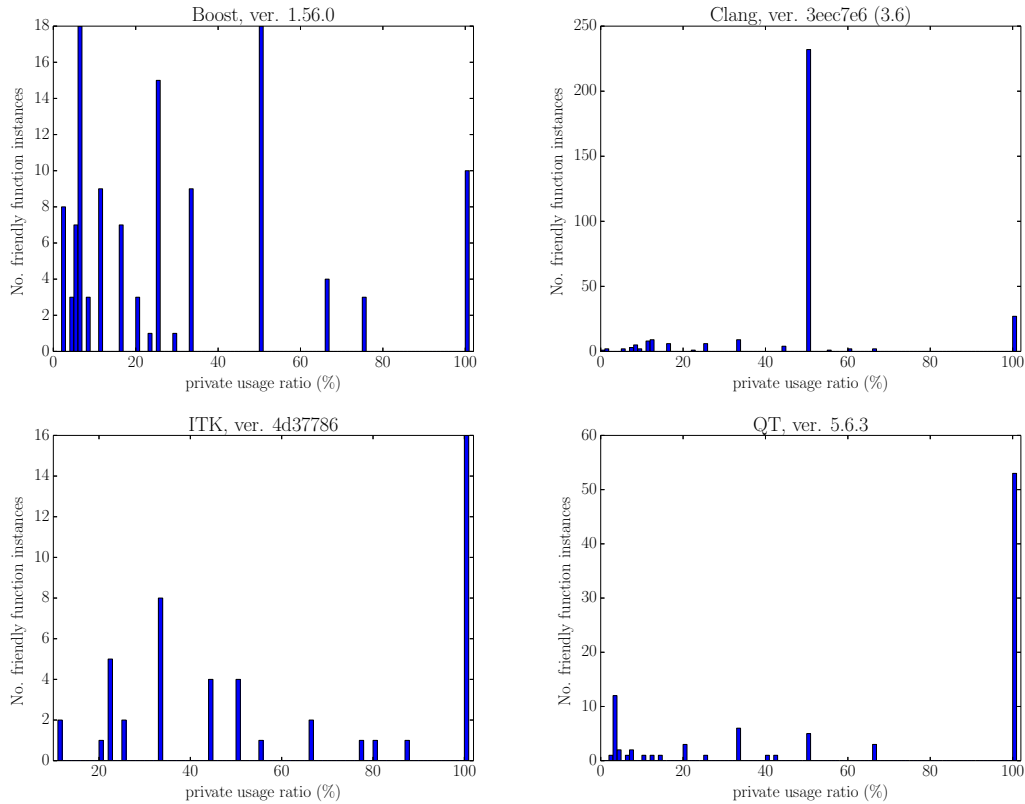
Figure 11. Private usage ratio in functions

access only a small subset of all the private members and this way the encapsulation is weakened. To overcome this issue, we propose a new language construct which enables friends to access only a restricted set of explicitly named private entities. Let us consider the syntax in Figure 13. During the compilation we would expect an error at the line when we attempt to access `a.y` private member, because we explicitly stated that `func` can access only `A::x`.

An alternative syntax conforming to the current C++17 standard is using attributes, shown in Figure 14. Even without any change to the current C++ standard, the attribute-based `friend_for` implementation could be useful. The tools (compilers and external tools) that support it could provide proper diagnostics when the semantics of the attribute is violated. The tools which do not support it would just simply ignore the `friend_for` specification. C++17 compliant compilers are required to silently ignore unknown attributes [46, 10.6.1/6], whereas C++14 compliant compilers may produce warnings [12, 7.6.1/5]. A proof-of-concept implementation using attributes is publicly available at [47].

4.1.1. Description of the Implementation The realization of the attribute [47] is based on the LLVM/Clang compiler infrastructure (version 3.6.0) [43]. The goal of this implementation is to prove that the idea is feasible, not to provide a full featured perfect realization. Therefore, we add some restrictions about the functionality and we do not implement proper error handling. We allow only functions to be declared as selective friends. We can select only one member to be the target of a selective friend. The argument of the attribute has to be a unary expression which can be parsed as a pointer to a member. Also, we use the GCC `__attribute__` syntax because the standard `[[attribute]]` syntax implementation is not complete in the used Clang version. Thus, the prototype handles the following syntax:

```
__attribute__((friend_for(&A::x))) friend void func(A &a);
```
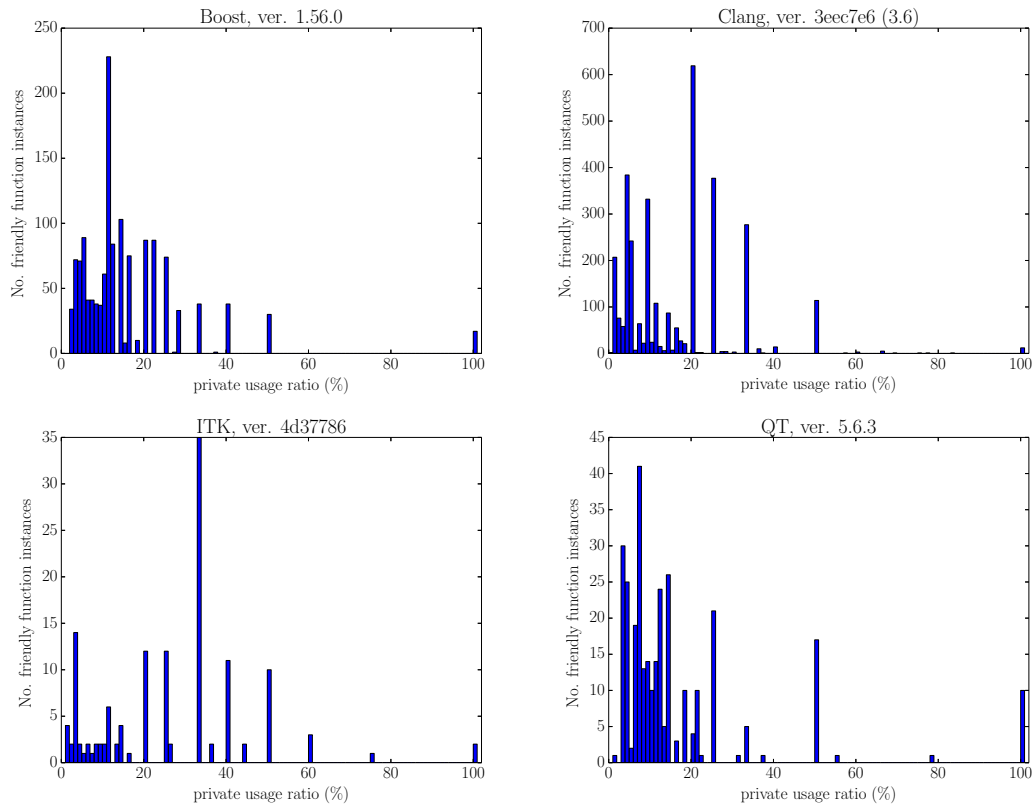
Figure 12. Private usage ratio in classes

```cpp
class A {
  int x = 0;
  int y = 0; // expected-note
             // {{implicitly declared private here}}
  friend for (x) void func(A &a);
};

void func(A &a) {
  a.x = 1; // OK
  a.y = 1; // expected-error
           // {{'y' is a private member of 'A'}}
}
```

Figure 13. Selective friend, a new lingual element

```cpp
[[friend_for(x)]] friend void func(A &a);

// Access of multiple members
[[friend_for(x, y)]] friend void func(A &a);
```

Figure 14. Attributes syntax for selective friends

We define our new attribute in Clang's `Attr.td` file:

```
def SelectiveFriend : InheritableAttr {
  let Spellings = [GCC<"friend_for">];
  let Args = [ExprArgument<"Expr">];
  let Subjects = SubjectList<[Function]>;
  let Documentation = [Undocumented]; }
```

`Spellings` defines the list of the supported attribute syntaxes, this time it is only the GCC style. The attribute syntax also defines the name of the attribute, in our case it is `"friend_for"`. `Args` specifies the list of the attribute arguments. Our attribute has only one argument which is an expression. This argument refers to a unary expression which takes the address of a member. `Subobjects` describes the list of the lingual elements that might have this attribute. In this case we allow only functions to have it. Note, there is no way to specify an attribute to be attachable only to friend function declarations, thus we had to be satisfied with the `Function` declarations as subjects.

Once we have the attribution definition in place then the Clang infrastructure will generate all the necessary parsing code. What left is to define the semantic action for the new attribute and to hook that action into the existent complier machinery. As for the hooking, we need to add a new function call in the `ProcessDeclAttribute` function:

```cpp
static void ProcessDeclAttribute(Sema &S, Scope *scope, Decl *D,
                                 const AttributeList &Attr,
                                 bool IncludeCXX11Attributes) {
  // ...
  case AttributeList::AT_SelectiveFriend:
    handleSelectiveFriendAttr(S, D, Attr);
    break;
  // ...
}
```

This function is dedicated to apply a specific attribute to the specified declaration if the attribute applies to declarations. (Our attribute applies to function declarations.)

The semantic action for the new attribute is defined as follows:

```cpp
1  static void handleSelectiveFriendAttr(
2      Sema & S, Decl * D,
3      const AttributeList &Attr) {
4
5    // TODO Add error handling, when D is not a
6    // FriendDecl
7
8    Expr *E = Attr.getArgAsExpr(0);
9
10   D->addAttr(
11       ::new (S.Context) SelectiveFriendAttr(
12           Attr.getRange(), S.Context, E,
13           Attr.getAttributeSpellingListIndex()));
14 }
```

The parameter `S` holds a reference of the monumental `Sema` class which is responsible for semantic analysis and AST building in the Clang compiler. The parameter `D` represents the declaration which has the attribute. The attribute itself is described with the `Attr` parameter. First, we retrieve the expression which is connected to the argument of the attribute (line 8). Note, we skip the handling of the erroneous case when the user attaches this attribute to a non-friend function. The second step is to create a `SelectiveFriendAttr` node in the AST (line 11). Then we register the new node to the function declaration with `addAttr`.

Clang has a semantic action associated for some of the production rules of the `postfix-expression` non-terminal in the C++ grammar [12, appendix 4]. These production rules are describing the syntax of a member access:

```
postfix-expression:
  postfix-expression . [template] id-expression
  postfix-expression -> [template] id-expression
```

In the associated semantic action, Clang checks whether the member access expression has privileges to access the actual member. In the process of checking the privileges, Clang eventually calls the `GetFriendKind()` function (Figure 15). This function iterates over all the friend declarations of the class that the member access expression refers to. The `S` parameter is a reference to the global `Sema` class. `EC` refers to the list of the enclosing functions and/or classes (up to the highest file scope) in which the member access expression is located. The `Target` parameter represents the member that the member access expression refers to. `Class` refers to the class that the member access expression deals with. This class is not necessarily a befriending class, it might

```
1   static AccessResult GetFriendKind(
2       Sema & S, const EffectiveContext &EC,
3       const AccessTarget &Target,
4       const CXXRecordDecl *Class) {
5
6     AccessResult OnFailure = AR_inaccessible;
7
8     // Okay, check friends.
9     for (auto *Friend : Class->friends()) {
10      switch (MatchesFriend(S, EC, Friend)) {
11      case AR_accessible:
12        return AR_accessible;
13
14      case AR_inaccessible:
15        continue;
16
17      case AR_dependent:
18        OnFailure = AR_dependent;
19        break;
20      }
21    }
22
23    // That's it, give up.
24    return OnFailure;
25  }
```

Figure 15. Checking the privileges in Clang with the GetFriendKind function

```
static AccessResult MatchesFriend(Sema &S, const EffectiveContext &EC,
                                  FunctionDecl *Friend) {
  // ...
  for (SmallVectorImpl<FunctionDecl *>::const_iterator
         I = EC.Functions.begin(),
         E = EC.Functions.end();
       I != E; ++I) {
    if (Friend == *I)
      return AR_accessible;
    // ...
  }
  // ...
}
```

Figure 16. MatchesFriend function: checking if the declarations are equal

not have any friend declarations at all. For each iteration (line 10), Clang checks for the iterated friend declaration whether any of the enclosing functions or classes (`EffectiveContext`) of the actually parsed member access expression is equal to that friend declaration (Figure 16). If that equality holds, that means the actual member access expression is valid, because one of the parent enclosing scopes is either a friend function or a friend class.

The point which we chose to intervene into Clang's original access checking mechanism is in the `GetFriendKind()` function as displayed in Figure 17. Whenever the `MatchesFriend()` function reports that the actual member is accessible then we review its decision by checking if there is a selective friend attribute (line 5-14).

Figure 18 shows how the constraint on the selective friend attribute is implemented. The `Friend` parameter refers to the friend declaration. The `Target` parameter represents the member that the actual member access expression refers to.

First (line 5-9), we check whether we can get a `NamedDecl` pointer from the friend declaration. If we cannot get such a pointer that means the friend declaration refers to a class, not a function. Since we do not handle classes we return with `AR_accessible`, i.e. we do not do any restriction.

Then we cast the `NamedDecl` into a `FunctionDecl`. If the cast is not successful that means either we deal with a function template or with a class template. If it turns out the friend declaration refers to a class template then again we leave intact the original access checking mechanism. Otherwise, if it is a function template then we get the pointer to the underlying `FunctionDecl` (line 11-22).

```
1   // Okay, check friends.
2   for (auto *Friend : Class->friends()) {
3     switch (MatchesFriend(S, EC, Friend)) {
4     case AR_accessible:
5       switch (SelectiveFriendConstraint(Friend,
6                                         Target)) {
7       case AR_accessible:
8         return AR_accessible;
9       case AR_inaccessible:
10        continue;
11      default:
12        assert(false &&
13               "should_not_reach_this_point");
14      }
15
16    case AR_inaccessible:
17      // ...
18    }
19  }
```

Figure 17. Intervention into Clang's access checking mechanism

```
1   static AccessResult SelectiveFriendConstraint(
2       FriendDecl * Friend,
3       const AccessTarget &Target) {
4
5     NamedDecl *ND = Friend->getFriendDecl();
6     // handling of friend classes not implemented
7     if (!ND) {
8       return AR_accessible;
9     }
10
11    FunctionDecl *FD = dyn_cast<FunctionDecl>(ND);
12    if (!FD) {
13      FunctionTemplateDecl *FTD =
14          dyn_cast<FunctionTemplateDecl>(ND);
15      // handling of friend class templates not
16      // implemented
17      if (!FTD) {
18        return AR_accessible;
19      }
20      FD = FTD->getTemplatedDecl();
21    }
22    assert(FD);
23
24    if (SelectiveFriendAttr *Attr =
25            FD->getAttr<SelectiveFriendAttr>()) {
26      const Expr *E = Attr->getExpr();
27      const UnaryOperator *UO =
28          cast<UnaryOperator>(E);
29      const DeclRefExpr *DRef =
30          cast<DeclRefExpr>(UO->getSubExpr());
31      if (cast<NamedDecl>(DRef->getDecl()) !=
32          Target.getTargetDecl()) {
33        return AR_inaccessible;
34      }
35    }
36
37    return AR_accessible;
38  }
```

Figure 18. Constraint on selective friend attribute

Finally (line 24-35), we retrieve the selective friend attribute from the function if it has any.
From the expression that is wrapped into the attribute we get the AST node that holds the
information about the unary operator expression. From the unary operator we get the AST node
(DeclRefExpr) that refers to the "address of" operator (&). Note, the used cast operations will
abort the program execution if the cast cannot succeed; this abort will not happen until we do casts
that are consistent with the attribution definition and its semantic action. Then we check whether

this referenced declaration (`DRef`) is equal to the declaration of the member that the member access expression refers to (`Target`). If they are not equal that means the actually investigated member access expression cannot have access to the member based on this specific friend declaration, therefore we return with `AR_inaccessible`; otherwise we return with `AR_accessible`.

By calling the `SelectiveFriendAttr()` function, we increase the complexity of the compilation process. To estimate the overhead, we did some measurements on the modified compiler. We generated a file which contains friend declarations and member access expressions. Let n denote the number of member accesses and x denote the number of friend declarations. For instance one generated file has the following form:

```
class A {
  int a;
  void mem_fun() {}
  friend void friend_fun0() {} // 1st
  friend void friend_fun1() {} // 2nd
  // ...
  friend void friend_funX() {} // xth
  friend void caller(A &a);
};
void caller(A &a) {
  [&a]() { a.mem_fun(); }(); // 1st
  [&a]() { a.mem_fun(); }(); // 2nd
  // ...
  [&a]() { a.mem_fun(); }(); // nth
}
```

In case of selective friends `class A` has a different form:

```
class A {
  int a;
  void mem_fun() {}
  __attribute__((friend_for(&X::x))) friend void friend_fun0() {} // 1st
  __attribute__((friend_for(&X::x))) friend void friend_fun1() {} // 2nd
  // ...
  __attribute__((friend_for(&X::x))) friend void friend_funX() {} // xth
  friend void caller(A &a);
};
```

We compiled the generated files several times, so we could measure the compilation time in the domain of n and x. We executed the compiler on an Intel(R) Core(TM) i7 class processor. Let us bind x to be a constant 100. Figure 19 displays the performance of the compiler, when the number of the friend functions is a constant (exactly 100) and the number of the member accesses is growing. `baseline` denotes the performance of the compiler without any modification, the `new` refers to the compiler with the modification in it and the `selective` denotes the performance of the modified compiler when we have the attached selective attributes to the friend declarations. Our conclusion is that the compile time of selective friends scarcely depend on the number of member access expressions when the number of the friend declarations is less than a hundred.

However, when we fix n to $1000(N)$ and let the number of the friend function declarations grow gradually, we get the results at Figure 20. Let $f(x)$ denote the compile time of N member accesses (`baseline`). Let $g(x)$ denote the compile time of N member accesses in the domain of the number of the friend declarations where those declarations are selective friends (`selective`). We can see a correlation between $f(x)$ and $g(x)$: $g(x) = C(x) * f(x), 1.12 < C(x) < 1.87, 1000 <= x <= 20000$. We state the compilation overhead is always less than 2x if the number of friend declarations are less than 20000.

We conclude that the performance of the compiler is indeed degraded, but we could measure noticeable compile time overhead only when the number of the friend declarations were above a hundred. (Note, we estimate that having more than a hundred friend declarations in a class is quite a rare case in C++ source codes.) The measurement scripts and the resulted files from which we generated the charts are publicly available online [47].

4.2. Eiffel like Syntax

In our implementation, we use expressions as attribute arguments: we are taking the address of a member variable. If we did the same with a member function that would certainly require the
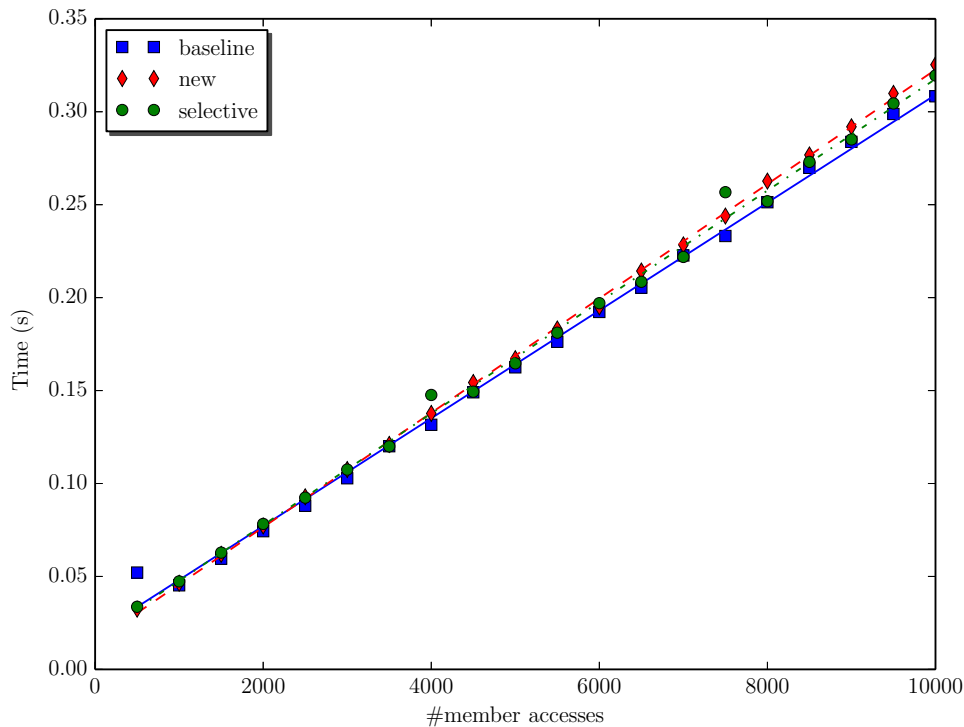
Figure 19. Comparing compile times, #friends: 100

compiler to generate the executable code of that function to be able to take its address. As a result, an otherwise non-generated (inlined) function is generated into the final executable. Also, in order to be able to take the address of a `static const` integral member variable, we need to define that variable [48, page 210–211]. So we would need to define it just because of the selective friend declaration. These side effects might not be wanted in some cases.

With our implementation, we have to specify a valid expression as an argument to the attribute. Therefore, we cannot restrict the access to a specific member type. For that we would need to have the attribute definition to look like this:

```
def SelectiveFriend : InheritableAttr {
  // ...
  let Args = [TypeArgument<"MemberType">];
  // ...
}
```

This is one limitation of the attributes implementation in the used Clang compiler, we cannot specify an argument which is either a type or an expression. We have to make the decision during the design of the attribute.

Because of the described difficulties, we might want to seek for other alternatives. One such alternative syntax could be to annotate each and every member via an attribute (Figure 21). The current C++ standard (C++17) enables us to tag the friend functions/classes which may access the private members. However, with the alternative approach presented in Figure 21 we would have to tag the members to indicate if they are accessible outside of the class. Actually, this method is very similar to Eiffel's access control.

The Eiffel programming language uses a special tagging mechanism for every class member to achieve selective friend like access control. Eiffel *features* are synonyms to C++ member variables and member functions. A feature can be either a field or a method [49, 50]. Features can be exported to any class [11].
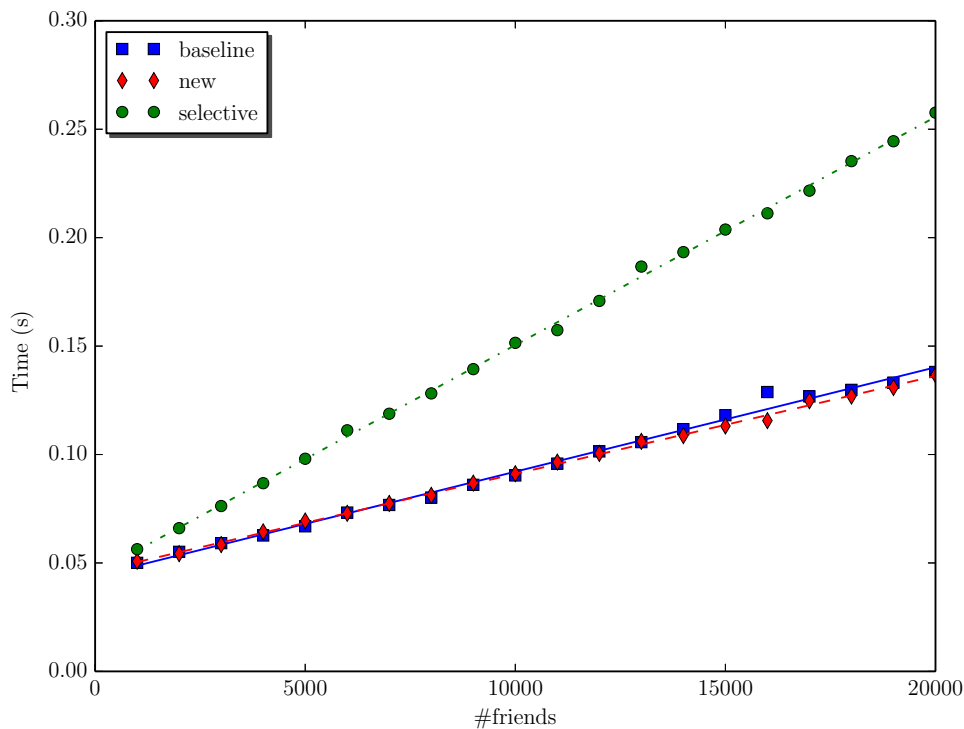
Figure 20. Comparing compile times, #members: 1000

```cpp
void func(A &a);
class A {
  [[friend_for (func)]] int x = 0;
  int y = 0;
};

void func(A &a) {
  a.x = 1; // OK
  a.y = 1; // expected-error,
           // {{'y' is a private member of 'A'}}
}
```

Figure 21. Alternative syntax: annotate the members to provide access for a function

```
feature {ANY}
```

The above means that the specific feature is available to all classes (analogous to C++ public).

```
feature {NONE}
```

The above means that the specific feature is not available to any classes (analogous to C++ private). We can enumerate concrete classes as well:

```
feature {CLASS_A, CLASS_B, CLASS_C}
```

In this case the features will be accessible by all three classes and in all of their proper descendant classes.

5. RELATED WORK

5.1. Private Usage of Friend Classes

In 2005, English, Buckley and Cahill defined a number of software metrics that measure the extent to which friend class relationships are actually used in systems [51]. Our experiments confirm these earlier results and extends them. They defined the following relevant metrics:

- Actual Friend Methods (AFM): Counts the number of methods in a friend class that access hidden members of the befriending class.
- Actual Friend Classes (AFC): The number of friend classes in a software system which are actually exploited through friendship, i.e. the number of those classes which have $AFM >= 1$. If we divide the AFC value of a system with the number of friend classes in it we may have percentage value about friend classes declared as friends superfluously.
- CCFF(1): Counts the number of distinct interactions of a friend class with hidden methods and attributes of the befriending class. (CCFF(1) is a refinement of the CCF(1) metric of Wilkie and Kitchenham [52].)
- RFFC(1): In a friend class, it counts the number of distinct hidden members which are accessed in the befriending class.
- CBOF(Back)): Counts the number of declared friend classes which actually access hidden members of the class, i.e. that utilize the friend declaration.
- Actual Friend Class Relationships (AFCR): This is a system level metric, like AFC. AFCR is the total sum of all the friend class relationships actually exploited in a system. Therefore, it is the sum of CBOF(Back) for all the classes in the system.
- Members Accessed by Friends (MAF): Counts the number of hidden members of a befriending class which are accessed by its friend classes.

All of these metrics are interpreted only in the context of friend classes, they are not defined for friend functions. Also, it is not clear how primary class templates, template instantiations and specializations affect these metrics. Our metrics of private usage (and private usage ratio) is applicable to friend functions and member functions of friend classes as well. Also, in our empirical study we handle class templates, their specializations and instantiations with a well defined algorithm as described in section 3.

English and Buckley have evaluated the above metrics on a number of opensource projects. For the systems in their study, the AFC percentage ranged from 42.8% to 100%. For about 60% of all systems the percentage of friend classes which exploit some of the friendship available to them was less than 75%. Therefore, for more than half of the systems in their study at least 25% of friend class declarations were shown to be redundant. The CBOF(Back) metric returned mostly small values. There were just 2 systems with a median value greater than 1 for CBOF(Back). In addition, the maximum value of CBOF(Back) was less than 9 for all but one system. This again indicated only a small subset of friend classes accessed hidden members. For the systems they analyzed, a large proportion of befriending classes returned a zero value for MAF, indicating that a considerable number of friend declarations were not exploited and as a result no hidden members of the befriending classes were accessed directly. In many systems a large proportion of classes had an RFFC(1) value of zero, indicating that no use was made of any friend relationship, where the class was declared as a friend. English and Buckley concluded there are many friend classes which do not exploit friendship. We confirm their conclusion based on our measurement results: The number of friend class function instances with zero private entity usage is quite high in the measured four libraries (Figure 10). With other words, there are many erroneous or superfluous friend declaration of friend classes.

They also concluded that there are many friend class declarations where friend function declarations might be more appropriate. This conclusion is also aligned with our empirical results in Figure 10 and 8. We can see that most of the friend class function instances have zero private usage. In case of friend functions the ratio of incorrect/correct friend function instances is significantly lower than the ratio in case of friend classes, thus friend function declarations are more appropriate.

There are some other results in English's study which support the raison d'etre of selective friends. The median value for the AFM metric was found to be 1 for many systems in their research. None of the 28 systems had a median value greater than 3. For 12 of the 28 systems, the median value of CCFF(1) was $<= 3$. They measured that in almost 50% of systems, 50% of the classes declared as friends were involved in less than or equal to 3 interactions that were dependent on the friend construct. Also, in all but 2 systems the median value of MAF is less than or equal to 3. The median value of RFFC(1) for 26 of the 28 systems was less than or equal to 3. Therefore, in almost all systems 50% of classes declared as friends required access to less than or equal to 3 hidden members in other classes. These numbers are aligned with our observation that the vast majority of the correct friend class function instances access only 1-4 private entities (Figure 10). English and Buckley concluded that for all systems at least 50% of classes declaring friends only access a small number of hidden members. We confirm this statement based on our metrics on private usage ratio of friend class function instances (Figure 12). They also concluded that the level of protection assigned to some class members should be reconsidered, especially where friend classes only access a small number of hidden members in a class. This again confirms the need for a selective friend construct.

5.2. Friends and Inheritance

Counsell and Newson studied a number of hypotheses about relationships between the use of friends and other internal attributes of a class [53]. Their results suggest that the friend mechanism is used as an alternative to inheritance. They examined four C++ systems of varying sizes and analysed data related to friends collected for each. The following five hypotheses were investigated:

- The more friends in a class, the less (non-friend) coupling found in that class.
- The more friends in a class, the more methods found in that class.
- Classes declared as friends of other classes have less inheritance than other system classes.
- Classes containing friends that engage in inheritance have fewer descendants than other system classes.
- Classes that do not engage in any inheritance have more friends than classes which do engage in inheritance.

Their results showed a lack of statistical significance with any of the class metrics collected. No evidence was found to support the hypothesis that classes declared as friends of other classes used inheritance any less than other classes. Strong evidence was found, however, to support the hypothesis that, firstly, friends tended to be found in classes deep in the inheritance hierarchy; secondly, that classes not engaging in inheritance use friends considerably more than classes that do. Their empirical evidence also suggests that friends are used primarily as a means for facilitating operator overloading.

English, Buckley and Cahill replicated and refined the empirical study made by Counsell and Newson [54, 55, 56, 57]. They refined their measurements by excluding friend constructs which participate in operator overloading. Counsell and Newson handled friendship symmetrically. However, classes or functions declared as friends have the potential to import additional functionality from classes which declare this class as a friend. Similarly, a class which declares friends has the potential to export more functionality to the classes or functions that it declares as friends. Therefore the coupling of classes is affected. Consequently, English and Buckley handled coupling metrics asymmetrically for the befriending classes and for the friend functions/classes.

Counsell and Newson included all overloaded operators (friend functions) in their measurements. English and Buckley excluded all overloaded friend operator functions. We neither exclude nor include all overloaded operators: we do not distinguish overloaded operators from regular functions. Although, we further refine and analyse the results with the concept of Meyers candidates (defined in Section 3.1.2). Meyers candidates are those friend functions which are using friendship not because of accessing private members but for other technical reasons (like automatic function template instantiation). An overloaded operator may or may not be a Meyers candidate, just as any other regular function. Meyers candidates influence our measurement results: We exclude Meyers

```cpp
class Key;

class Owner {
public:
  void privileged(const Key &) {}
private:
  void foo();
};
class Key {
  friend class User;
  friend void userFunc(Owner &owner);
private:
  Key() {}
};
class User {
public:
  void usePrivileged(Owner &owner) {
    owner.privileged(Key{}); // OK
  }
};
void userFunc(Owner &owner) {
  owner.privileged(Key{}); // OK
}
void noAccess(Owner &owner) {
  owner.privileged(Key{}); // compile error
}
```

Figure 22. The access key idiom

candidates in the calculation of the private usage ratio. We do not tag friend functions as being erroneously friends if they are Meyers candidates.

5.3. Alternatives for Selective Friends

With current C++ we can achieve semantically similar effects to the proposed selective friend language construct. However they all have their own drawbacks and difficulties.

5.3.1. The Access Key Idiom First, there is no such acknowledged pattern which has this name. Some people tend to refer the presented methodology as key-oriented access protection, passkey idiom [58], access key idiom or higher-order friendship [59]. In this paper we refer this as the *access key idiom*. In Figure 22, the Owner class has a privileged function, of which we want to restrict the access. This is achieved by the fact that this function requires a Key object to be passed as an argument. Since Key has a private constructor only its friends can construct any object from it. Therefore Key controls which classes and functions can have access to the privileged function. With this idiom we can provide selective friendship: we can control access to a specific member function and we can restrict access of friends to a set of private methods. (Owner::foo cannot be accessed by anybody). The drawback of this pattern is that we cannot handle the accessing of member fields with it, we can handle only member functions.

5.3.2. The Attorney-Client Idiom This pattern uses an auxiliary class to handle (restrict) the access to the private members of a specific class. This intermediate class is the Attorney; just as in real life it protects its Client, so other entities can access it only via the lawyer [60]. In Figure 23 the Bar class has a limited access to Client, it can call only Client::A() via Attorney::callA(). So Bar acts as a selective friend of class Client. Also, the Attorney controls which classes can have access to the internals of Client. Attorney has a similar role as Key has in the access key idiom. The use of this idiom does not scale well, because we would need to define several attorney classes if we wanted to provide access for the different combination of members. Also, using too many attorneys might result in unmaintainable and hardly understandable code.

```cpp
class Client
{
private:
  void A(int a);
  void B(float b);
  void C(double c);
  friend class Attorney;
};

class Attorney {
private:
  static void callA(Client & c, int a) {
    c.A(a);
  }
  friend class Bar;
};

class Bar { /* ... */  };
```

Figure 23. The Attorney-Client Idiom

6. FUTURE RESEARCH

It is an important candidate for future work to create a selective friend implementation without the current limitations. For instance we could enhance the implementation for selective friend classes, or we could further develop it to support more than one selected member. Also, we are planning to implement this new language element without attributes.

It is worth to investigate the possibilities of `const` qualified selective friends. For instance, consider the following code block:

```cpp
class A {
  int x = 0;
  int y = 0;
  friend for (x) B const; // read-only
};
void B::func(A &a) { // expected-error, B has const access only
  int i = a.x;
  a.x = 1; // or expected-error here, a is implicitly casted to 'const A&'
}
```

The member functions of B could access only the listed members of A and every instance of A would be handled as a `const` object.

7. CONCLUSION

We executed a fine-grained measurement about the usage of friends on several open source projects. Based on these empirical results we claim that friendship in C++ is often misused. In a number of cases friend functions access only public members or not access members at all. In other cases they gain superfluous access to private members, which is a possible source of errors. Current C++ compilers and static analysis tools do not issue a warning on suspicious friend usage. However, with our publicly available friend inspection tool we can list the possible erroneous uses of friend declarations.

The current C++ language specification does not allow to restrict the access of private members. With selective friends, we can establish a more sophisticated access control. Our proposal may decrease the degradation of encapsulation. We created a proof-of-concept implementation based on the LLVM/Clang compiler infrastructure to show that such constructs can be established with a minimal syntactical and compilation overhead. Even without any change to the current C++ standard, the attribute-based `friend_for` implementation could be useful. The compilers that support it could provide proper diagnostics when the semantics of the attribute is violated. The tools which do not support it would just simply ignore the `friend_for` specification as unknown attributes must be ignored according to C++17.

REFERENCES

1. Nierstrasz O. A survey of object-oriented concepts. *Object-Oriented Concepts, Databases, and Applications*, Kim W, Lochovsky FH (eds.). ACM PRESS: New York, 1989; 3–21.
2. Snyder A. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Not.* 1986; **21**(11):38–45.
3. Schärli N. Composable encapsulation policies. ECOOP '04, 2004; 248–274.
4. Hogg J. Islands: Aliasing protection in object-oriented languages. *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, ACM: New York, NY, USA, 1991; 271–285, doi:10.1145/117954.117975. URL http://doi.acm.org/10.1145/117954.117975.
5. Almeida PS. Balloon types: Controlling sharing of state in data types. *Proceedings ECOOP'97*, *LNCS*, vol. 1241, Springer-Verlag, 1997; 32–59.
6. Noble J, Vitek J, Potter J. Flexible alias protection. *Proceedings ECOOP'98*, LNCS, 1998; 158–185.
7. Aldrich J, Kostadinov V, Chambers C. Alias annotations for program understanding. *SIGPLAN Not.* Nov 2002; **37**(11):311–330, doi:10.1145/583854.582448. URL http://doi.acm.org/10.1145/583854.582448.
8. Kniesel G, Theisen D. JAC - Java with transitive readonly access control. *Proceedings of the Intercontinental Workshop on Aliasing in Object-Oriented Systems*, Lisbon, Portugal, 1999.
9. Boyapati C, Liskov B, Shrira L. Ownership types for object encapsulation. *SIGPLAN Not.* Jan 2003; **38**(1):213–223, doi:10.1145/640128.604156. URL http://doi.acm.org/10.1145/640128.604156.
10. Schärli N, Black AP, Ducasse S. Object-oriented encapsulation for dynamically typed languages. *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, ACM: New York, NY, USA, 2004; 130–149, doi:10.1145/1028976.1028988. URL http://doi.acm.org/10.1145/1028976.1028988.
11. Eiffel Documentation. Adding class features, making features available to clients 2016. URL https://docs.eiffel.com/book/platform-specifics/adding-class-features#Making_Features_Available_to_Clients, accessed: 2016-06-12.
12. ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization: Geneva, Switzerland, 2014.
13. isocpp org. C++ FAQ, Friends 2016. URL https://isocpp.org/wiki/faq/friends, accessed: 2016-06-12.
14. isocpp org. C++ FAQ, Friends, Should my class declare a member function or a friend function? 2016. URL https://isocpp.org/wiki/faq/friends#member-vs-friend-fns, accessed: 2016-06-12.
15. Stroustrup B. *The C++ programming language*. Pearson Education, 2013.
16. Stroustrup B. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co.: New York, NY, USA, 1994. P. 53.
17. Meyers S. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005. Item 46, p. 256.
18. Sutter H, Alexandrescu A. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.
19. Bolton AR. Friendship and the attorney-client idiom January 2006. URL http://www.drdobbs.com/friendship-and-the-attorney-client-idiom/184402053, accessed: 2018-02-22.
20. Misfeldt T, Bumgardner G, Gray A, Xiaoping L. *The Elements of C++ Style*. Cambridge University Press, 2004. P. 77.
21. Josuttis N. *Object-Oriented Programming in C++*. Wiley, 2002.
22. Oracle. Controlling access to members of a class 2015. URL https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html, accessed: 2016-06-12.
23. Nidhra S, Dondeti J. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)* 2012; **2**(2):29–50.
24. Khan ME, Khan F, *et al.*. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl* 2012; **3**(6).
25. Stackoverflow. Is there a way to simulate the c++ 'friend' concept in java? 2008. URL http://stackoverflow.com/questions/182278/is-there-a-way-to-simulate-the-c-friend-concept-in-java, accessed: 2016-06-12.
26. Stackoverflow. Why can outer java classes access inner class private members? 2009. URL http://stackoverflow.com/questions/1801718/why-can-outer-java-classes-access-inner-class-private-members, accessed: 2016-06-12.
27. Gosling J, Joy B, Steele G, Bracha G, Buckley A. The java language specification, java se 7 edition, determining accessibility 2016. URL http://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html#jls-6.6.1, accessed: 2016-06-12.
28. Microsoft Developer Network. Access modifiers (c# reference) 2016. URL https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx, accessed: 2016-06-12.
29. Stackoverflow. What is the c# equivalent of friend? 2008. URL http://stackoverflow.com/questions/204739/what-is-the-c-sharp-equivalent-of-friend, accessed: 2016-06-12.
30. Microsoft Developer Network. Internalsvisibletoattribute class 2016. URL https://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.internalsvisibletoattribute.aspx, accessed: 2016-06-12.
31. Dlang org. D programming language, friends 2016. URL http://dlang.org/cpptod.html#friends, accessed: 2016-06-12.
32. Matsakis ND, Klock II FS. The rust language. *ACM SIGAda Ada Letters*, vol. 34, ACM, 2014; 103–104.
33. The Rust Project Developers. Rust documentation 2018. URL https://doc.rust-lang.org/, accessed: 2018-02-20.

34. Python Software Foundation. The python language reference 2016. URL https://docs.python.org/2/reference/, accessed: 2016-06-12.
35. Comprehensive Perl Archive Network. perltoot - tom's object-oriented tutorial for perl 2018. URL http://search.cpan.org/dist/perl-5.8.9/pod/perltoot.pod, accessed: 2018-02-20.
36. Savikko V. Design patterns in python. *Proceedings of the 6th International Python Conference*, Citeseer, 1997.
37. LLVM org. Clang 3.7 documentation, tutorial for building tools using libtooling and libastmatchers 2016. URL http://clang.llvm.org/docs/LibASTMatchersTutorial.html, accessed: 2016-06-12.
38. Márton G. Friend statistics 2016. URL https://github.com/martong/friend-stats, accessed: 2016-06-12.
39. Boost org. Boost operators library 2008. URL http://www.boost.org/doc/libs/1_56_0/libs/utility/operators.htm, accessed: 2017-04-18.
40. Coplien JO. Curiously recurring template patterns. *C++ Rep.* Feb 1995; **7**(2):24–27. URL http://dl.acm.org/citation.cfm?id=229227.229229.
41. Schling B. *The Boost C++ Libraries*. XML Press, 2011. URL https://theboostcpplibraries.com/.
42. Boost org. Boost c++ libraries 2016. URL http://www.boost.org/, accessed: 2016-06-12.
43. LLVM org. clang: a c language family frontend for llvm 2016. URL http://clang.llvm.org, accessed: 2016-06-12.
44. US National Library of Medicine of the National Institutes of Health. Insight segmentation and registration toolkit (itk) 2016. URL http://www.itk.org/, accessed: 2016-06-12.
45. Qt. Cross-platform software development for embedded & desktop 2018. URL https://www.qt.io/, accessed: 2018-01-30.
46. ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization: Geneva, Switzerland, 2017.
47. Márton G. Selective friend 2016. URL https://github.com/martong/clang/tree/selective_friend, accessed: 2016-06-12.
48. Meyers S. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. 1st edn., O'Reilly Media, Inc., 2014.
49. ISO. *ISO/IEC 25436:2006 - Eiffel: Analysis, Design and Programming Language*. International Organization for Standardization: Geneva, Switzerland, 2006.
50. wikpedia org. Eiffel (programming language) 2016. URL http://en.wikipedia.org/wiki/Eiffel_(programming_language), accessed: 2016-06-12.
51. English M, Buckley J, Cahill T. A friend in need is a friend indeed [software metrics and friend functions]. *Empirical Software Engineering, 2005. 2005 International Symposium on*, IEEE, 2005; 10–pp.
52. Wilkie FG, Kitchenham B. An investigation of coupling, reuse and maintenance in a commercial c++ application. *Information and Software Technology* 2001; **43**(13):801–812.
53. Counsell S, Newson P. Use of friends in c++ software: an empirical investigation. *Journal of Systems and Software* 2000; **53**(1):15–21.
54. English M, Buckley J, Cahill T, Lynch T. An empirical study of the use of friends in c++ software. *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, IEEE, 2005; 329–332.
55. English M, Buckley J, Cahill T, Lynch K, English M. An analysis of the use of friends in c++ software systems ; .
56. English M, Buckley J, Cahill T. A replicated and refined empirical study of the use of friends in c++ software. *Journal of Systems and Software* 2010; **83**(11):2275–2286.
57. English M, Cahill T, Buckley J. Construct specific coupling measurement for c++ software. *Computer Languages, Systems & Structures* 2012; **38**(4):300–319.
58. Stackoverflow. How to name this key-oriented access-protection pattern? 2010. URL http://stackoverflow.com/questions/3324248/how-to-name-this-key-oriented-access-protection-pattern, accessed: 2016-06-12.
59. Bergé A. Tales of c++, friends with benefits 2013. URL http://talesofcpp.fusionfenix.com/post-4/episode-three-friends-with-benefits, accessed: 2016-06-12.
60. wikibooks org. More c++ idioms, friendship and the attorney-client 2014. URL http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Friendship_and_the_Attorney-Client, accessed: 2016-06-12.