Date: 10/10/2014
Reply to: Andrea Proli
(andy.prowl at gmail dot com)

Dynamic Generic Programming with Virtual Concepts

Author: Andrea Proli

Abstract

The present proposal aims at introducing language support for type erasure in C++ in the form of so-called *virtual concepts*. Virtual concepts allow extending the set of interfaces a type adheres to without modifying its definition, and without requiring generic algorithms and data types that rely on those interfaces to be written as templates.

The proposed solution works both for user-defined types and for fundamental types, and does not require user-defined types to be polymorphic – i.e. they may have no virtual member functions. Adherence of a type to a concept can be either deduced (through *duck-typing*) or asserted (through *concept maps*).

Both value semantics and reference semantics are supported. Supporting value semantics means allowing the manipulation of erased types through the typical operations on value types (assignment, copy- and move-construction, equality comparison, etc.); supporting reference semantics means allowing the manipulation of erased types through native references and pointers, without requiring any sort of proxy or wrapper.

In terms of efficiency, the proposed solution yields *in the worst case* the same overhead as inheritance-based polymorphism for the use cases that the latter supports – namely, reference semantics – both memory-wise and performance-wise.

Virtual concepts offer a simple alternative to type erasure techniques which are usually hard to implement, limited in expressivity, and/or make code hard to read and slow to build. In particular, virtual concepts simplify and generalize common library utilities such as `std::any` and `std::function`.

Finally, virtual concepts provide a native and elegant solution to the Expression Problem [1] and to the Proxy Dilemma [2] in C++.

Table of Contents

Table of Figures

1. Introduction

Generic Programming (GP) is a programming paradigm which encourages the formulation of algorithms in terms of a minimal set of abstract requirements on the types of the objects they operate on. This set of requirements, which is said to define a *concept*, takes the role of a contract regulating the interaction between the generic algorithm and its inputs.

At a *logical* level, concepts represent the layer of indirection which allows separating high-level, abstract operations from type-specific implementation details, breaking the dependency of the former on the latter by making both depend on an intermediate entity:

Figure 1 – Abstraction role of concepts in Generic Programming

This logical layer of indirection makes it possible to modify the behavior of an algorithm by substituting any of the concrete types it operates on for a different type *without requiring any change to the definition of the algorithm itself*, as long as the new types satisfies the corresponding concept. The generic algorithm becomes therefore a piece of highly reusable software.

In C++, GP is most often achieved through the use of templates. Templates offer a mechanism for parameterizing algorithms (i.e. function templates) and data structures (i.e. class templates) with type and non-type arguments that are known *at compile-time* – which is why this mechanism is commonly referred to as *static polymorphism*.

The main benefits of using templates for GP are:

1. *Composability*: The modeling relation between a type T and a concept C is *deduced* by the compiler based on syntactic conformance of T's interface to C's requirements; conformance is not asserted on T's definition. Consequently, making T a model of a *new* concept D will not require changing the definition of T;
2. *Value semantics*: Unlike inheritance, templates do not require generic algorithms to access their arguments through pointers or references. This is in line with C++'s philosophy of supporting user-defined types that "behave like an `int`". Reference semantics is, however, not precluded;
3. *Efficiency*: Since templates are instantiated at compile-time, and the definition of type arguments must be made available to the compiler when instantiating a generic algorithm, the compiler has enough information to inline function calls inside the algorithm's body; this results in machine code which is often as fast as (or even faster than) the one produced by a hand-written, non-generic algorithm.

The last point is a double-edged sword though: requiring an argument type T's definition to be available at compile-time to the generic algorithm means that (1) changes to T's definition will require recompiling all the translation units which provide objects of type T as an input to a generic algorithm (including the algorithm's code) and (2) the algorithm's definition cannot be kept in a separate translation unit. At a *physical* level, the independence schematized in Figure 1 no longer holds:

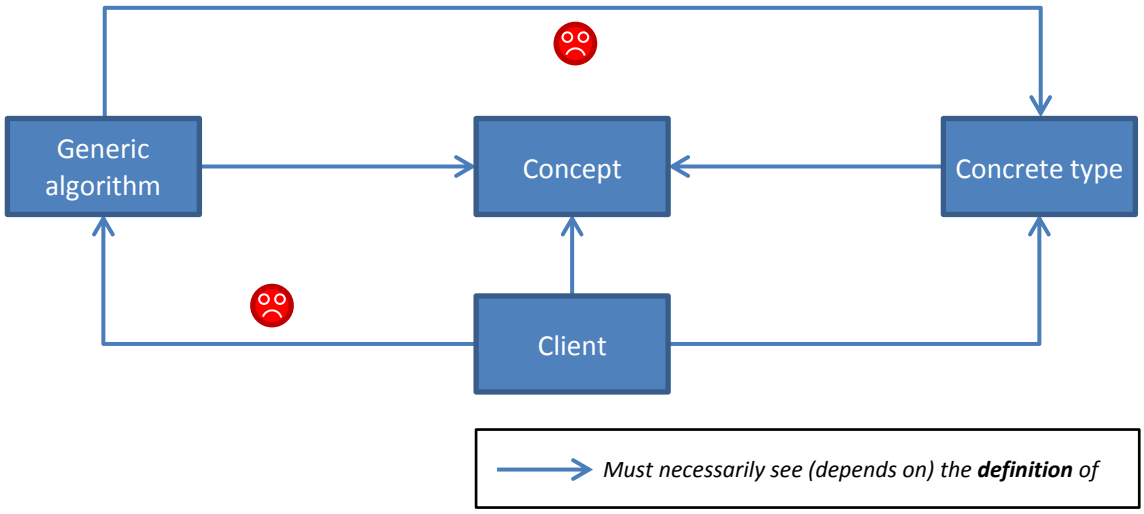

Figure 2 – Physical dependency schema for static polymorphism based on templates

If independent compilation and deployment is important, the programmer is forced to resort to alternative techniques. The most popular among such techniques in OOP languages is inheritance-based *dependency inversion* [3], which allows execution control to flow from the generic algorithm to the concrete type at run-time without the former being physically dependent on the latter:

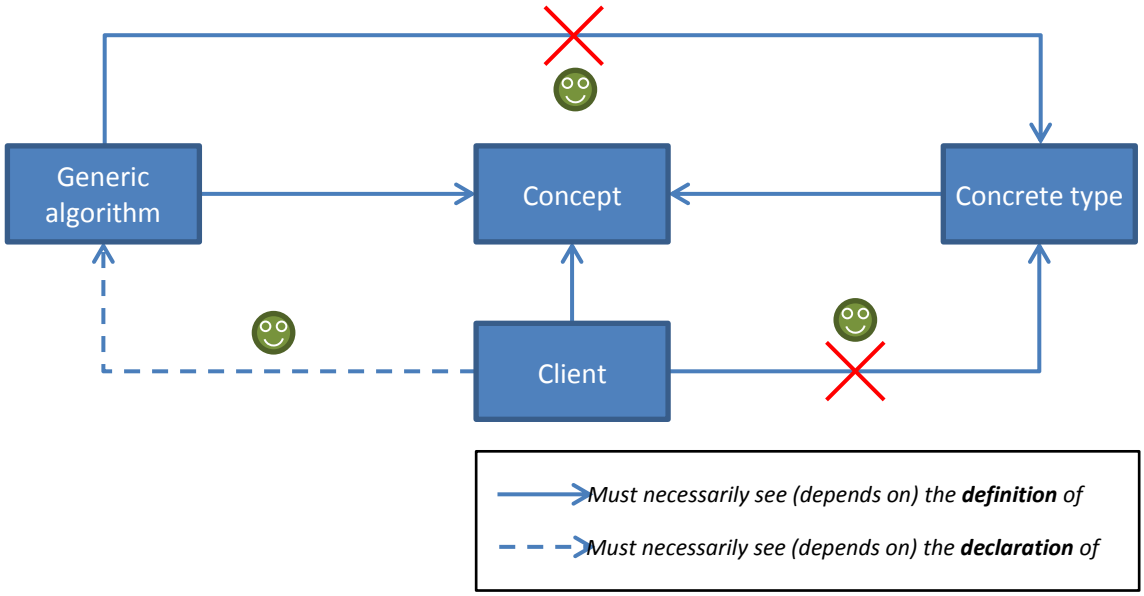

Figure 3 – Physical dependency schema for dynamic polymorphism based on inheritance

While supporting separate compilation, inheritance-based polymorphism has several shortcomings:

1. *Intrusiveness*: the adherence of a type to a contract must be mentioned in that type's definition. This is a problem for extensibility: if we wanted to make an existing type adhere to a *new* contract in order to enable polymorphic dispatch, we would have to change that type's definition – which may be troublesome or not possible;
2. *Applicability*: inheritance-based polymorphism is limited to user-defined types, because the language forbids making a fundamental type (e.g. `int`) derive from an abstract class to implement its interface. This means we cannot provide an object of a fundamental type to a generic algorithm defined in terms of an abstract interface;
3. *Reference semantics*: inheritance-based polymorphism forces on the programmer the use of pointers or references to access objects of polymorphic types. While value semantics can be achieved, the adoption of sophisticated techniques is inevitable [4] [5];
4. *Performance*: Run-time function call binding prevents the compiler from performing precious optimizations such as function call inlining.

Of all these drawbacks, only the last one is inherent to dynamic polymorphism *as a pattern*: the previous ones are characteristic of inheritance *as a technique*.

This proposal aims at introducing language support for type erasure in C++ that would allow for a run-time form of GP – which we refer to as *Dynamic Generic Programming* (DGP) – in a way similar to how templates and concepts [6] support compile-time GP.

The proposed solution, based on what we call *virtual concepts*, provides the composability and flexibility of classical compile-time GP, while retaining the advantages of inheritance in terms of independent deployability. The price to be paid in exchange for these benefits is the inherent performance penalty of dynamic binding, but no more than that – see Section 3 for a detailed account on our design goals.

The rest of this document is structured as follows: Section 2 provides an overview of the current state-of-the-art approaches to DGP in C++; Section 3 states the design goals which underlie and shape our proposal; Section 4 gives a brief demonstration of the capabilities of virtual concepts, showing how they overcome the limitations of existing DGP techniques; Section 5 shows how to define a virtual concept, and how to extract a type-erasing interface definition from the associated requirements; Section 6 covers the details of how virtual concepts support value semantics and reference semantics; Section 7 illustrates the mechanisms of *duck-typing* and *concept maps* to express the modeling relationship between a type and a virtual concept; Section 8 introduces the fundamental relationships between concepts, such as *refinement* and *subsumption*, and discusses their decidability; Section 9 elaborates on the interplay between virtual concepts, templates, and inheritance; Section 10 focuses on the modifications required to C++'s overload resolution algorithm; Section 11 explains how to revert the type-erasing procedure through *concept casting*; Section 12 presents a possible implementation technique and reports on implementation experience; finally, Section 13 draw the conclusions.

2. Existing approaches to DGP

The C++ Standard Library provides types such as `std::function`, for type-erasing callable objects compatible with a given signature, and `std::any`, for type-erasing objects of any type; `std::shared_ptr` offers support for type-erasing custom deleters. However, neither the Standard Library nor the core language provides general-purpose tools for defining new type-erasing wrappers.

Several attempts have been made at filling this gap at a library level. An excellent overview of type-erasure techniques is given in [7] (parts I through IV), including a comparison between two of the most popular publicly-available library solutions: Adobe.Poly [8] and Boost.TypeErasure [5].

An interesting recent work is Pyry Jahkola's C++11 Poly library [9], which is nevertheless not meant for production use. Also worth mentioning is the Boost.Interfaces library [10] (which is *not* part of Boost [11] and seems to be no longer maintained) as one of the earliest attempts to support DGP in C++.

A radically different approach, addressing the problem at a language level, was chosen for the Signatures project [12], implemented as a language extension for GCC (up to version 2.95) [13] [14].

We shall now set up a working example that will help us evaluate and compare these solutions.

2.1. Working example

We define two classes, `Circle` and `Rectangle`, as shown below. Both classes have member functions `get_area()` and `get_perimeter()` that return a `double`. These classes do not derive from any common base class, and we will assume them to be part of a closed-source, third-party static library we are linking into our project. This implies that we are not allowed to modify their definitions:

```
struct Circle
{
    Circle();
    Circle(Point const& center, double radius);
    double get_area() const;
    double get_perimeter() const;
private:
    // ...
};

struct Rectangle
{
    Rectangle();
    Rectangle(Point const& center, double length, double width);
    double get_area() const;
    double get_perimeter() const;
private:
    // ...
};
```

Our goal is to support the definition of the generic algorithm `print_areas()` shown below:

```
void print_areas(std::vector<Shape> const& v)
{
    for (auto const& s : v)
    {
        std::cout << s.get_area() << " ";
    }
}
```

`print_areas()` accepts in input a heterogenous vector of `Shape` objects, where `Shape` is some erasing wrapper that is capable of representing any object whose type satisfies the associated concept's requirements.

For our purposes, the concept of a *shape* requires a conforming type to offer non-mutating functions `get_area()` and `get_perimeter()` that return, respectively, the area and the perimeter of the object. It is easy to verify that both classes `Circle` and `Rectangle` satisfy these requirements.

Here is how the requirements could be expressed using the syntax proposed by the Concepts TS [6]:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
    requires SemiRegular<T>;
};
```

The obvious constraint that makes this problem hard is that the translation unit in which `print_areas()` is defined may not include the header files containing the definitions of `Circle` and `Rectangle`; yet, `print_areas()` must be able to dispatch function calls to instances of those types through the `Shape` wrapper.

Furthermore, we expect the program below to be well-formed and to yield the correct output (assuming all the necessary object files are linked in, and omitting standard header inclusions):

```
#include "shape.hpp" // CONTAINS THE DEFINITION OF THE TYPE-ERASING WRAPPER

void print_areas(std::vector<Shape> const& v);

int main()
{
    auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

    auto c = Circle{{0.0, 0.0}, 42.0};

    std::vector<Shape> v{Shape{r}, Shape{c}};

    print_areas(v);
}
```

2.2. Adobe.Poly

Adobe.Poly [8] is part of the open-source Adobe Source Libraries (ASL) [15]. The goal of Adobe.Poly is to allow "*creating regular, runtime-polymorphic objects with value semantics*" [15]. The philosophy and theoretical underpinnings of Adobe.Poly are discussed in detail in [4], [16], [17], and [18]. The source code for ASL can be found at http://sourceforge.net/projects/adobe-source/.

Adobe.Poly, which predates the Concepts TS, does not allow generating type-erasing wrappers directly from concept definitions like the one given in Section 2.1. Instead, it requires the user to define (for each concept) one abstract interface, one class template, and one concrete class; utilities provided by the framework help reducing the amount of boilerplate code that the user is required to type.

We will start with the abstract interface, which is *not* meant to be used directly by generic algorithms like `print_areas()`; instead, it is used internally for dispatching function calls to the erased types. It defines the common operations that the types satisfying our concept must support, but does not require their signatures to match exactly those in the erased types, nor to match those of the `Shape` concept; not even a one-to-one correspondence between these functions is necessary, albeit common.

Here is how we could define our `ShapeInternal` interface:

```
struct ShapeInternal : adobe::poly_copyable_interface
{
    virtual double get_area_internal() const = 0;
    virtual double get_perimeter_internal() const = 0;
};
```

The abstract base class `poly_copyable_interface` is part of ASL, and defines common operations on copyable types as well as other functions used by the framework.

Next, we need to define a generic holder for our type-erased objects. The holder is parameterized by the type of the wrapped object, and implements `ShapeInternal` by delegating to it:

```
template<typename T>
struct ShapeHolder : adobe::optimized_storage_type<T, ShapeInternal>::type
{
    using base =
          typename adobe::optimized_storage_type<T, ShapeInternal>::type;

    ShapeHolder(adobe::move_from<ShapeHolder> x)
        : base{adobe::move_from<base>{x.source}} { }

    ShapeHolder(T x) : base_t{std::move(x)} { }

    virtual double get_area_internal() const override
    { return this->get().get_area(); }

    virtual double get_perimeter_internal() const override
    { return this->get().get_perimeter(); }
};
```

The relevant portions of the definition of `ShapeHolder` are the constructor accepting an object of type T, which eventually gets stored by the holder's base constructor, and the delegating implementation of `get_area_internal()` and `get_perimeter_internal()` – everything else is for integration with the framework. Also notice, that the base class template `optimized_storage_type` – which implements the Small Buffer Optimization (SBO) – must be instantiated with the type of the abstract interface (`ShapeInternal`).

Finally, the concrete class: this is the type which defines the interface of the concept, i.e. the operations that generic algorithms can invoke on the erased objects. We will call ours `ShapeEraser`. The actual erasing wrapper used by generic algorithms like `print_areas()` can be obtained by instantiating the `adobe::poly` class template with the eraser type:

```
struct ShapeEraser : adobe::poly_base<ShapeInternal, ShapeHolder>
{
    using base = adobe::poly_base<ShapeInternal, ShapeHolder>;

    using base::base; // INHERIT BASE CLASS'S CONSTRUCTORS

    ShapeEraser(adobe::move_from<ShapeEraser> x)
        : base_t{adobe::move_from<base_t>(x.source)} { }

    double get_area() const
    { return interface_ref().get_area_internal(); }

    double get_perimeter() const
    { return interface_ref().get_perimeter_internal(); }
};

using Shape = adobe::poly<ErasedShape>;
```

With all of this in place, the test code from Section 2.1 can compile and run correctly.

2.2.1. Adaptation vs. duck-typing

In the working example from Section 2.1, classes `Circle` and `Rectangle` have a syntactically uniform interface: in other words, the operations which return the area and perimeter of an object are realized by member functions with identical signature.

This made our life simpler when implementing the dispatch mechanism, because it allowed us to write a single class template (`ShapeHolder`) for delegating the implementation of those logical operations to any of our concrete types.

Had we attempted to use a `Shape` wrapper to type-erase an object with differently-named member functions, the result would have been a (likely cryptic) compiler error.

Consider, for instance, the following `Triangle` class, whose functions `calculate_area()` and `calculate_perimeter()` do not keep up with our current `ShapeHolder`'s expectations:

```
struct Triangle
{
    Triangle(Point const& v1, Point const& v2, Point const& v3);
    double calculate_area() const; // NON-CONFORMING NAME!
    double calculate_perimeter() const; // NON-CONFORMING NAME!
private:
    Point _v1;
    Point _v2;
    Point _v3;
};
```

Without further intervention, the code below would fail to compile:

```
auto t = Triangle{{0.0, 0.0}, {1.0, 0.0}, {0.0, 1.0}};

Shape s = t; // ERROR!
```

The approach to determining whether a type models a concept based exclusively on syntactic conformance is called *duck-typing*. Duck-typing has advantages and disadvantages depending on the use case, but Adobe.Poly allows performing type-specific adaptation by specializing the `ShapeHolder` class template for concrete types:

```
template<>
struct ShapeHolder<Circle>
    : adobe::optimized_storage_type<Circle, ShapeInternal>::type
{
    using base_t =
          typename adobe::optimized_storage_type<Circle, ShapeInternal>::type;

    ShapeHolder(Circle x) : base_t{std::move(x)} { }

    ShapeHolder(adobe::move_from<ShapeHolder> x)
        : base_t{adobe::move_from<base_t>{x.source}} { }

    virtual double get_area_internal() const override
    { return this->get().calculate_area(); }

    virtual double get_perimeter_internal() const override
    { return this->get().calculate_perimeter(); }
};
```

Adaptation provides a lot of flexibility when expressing the modeling relation between a type and a concept: in particular, types can be freely designed without worrying about the (possibly conflicting) syntactic requirements of all the generic algorithms that may use them.

2.2.2. Reference semantics

Adobe.Poly was designed to support GDP for types used with value semantics, and does not offer any support for reference semantics off-the-shelf. However, reference semantics use cases can be covered with some additional adaptation work.

In particular, to support reference semantics for a given concept, it is possible to define a new holder class template for instances of `std::reference_wrapper`, with proper adaptation performed in the function calls that delegate to the wrapped object. In the case of `ShapeHolderRef` below, the delegating functions `get_area_internal()` and `get_perimeter_internal()` would have to be rewritten so that `reference_wrapper`'s `get()` member function is called to access the object:

```
template<typename T>
struct ShapeHolderRef; // INTENTIONALLY LEFT UNDEFINED

template<typename T>
struct ShapeHolderRef<std::reference_wrapper<T>>
    : adobe::optimized_storage_type<std::reference_wrapper<T>,
                                    ShapeInternal>::type
{
    using base_t = typename adobe::optimized_storage_type<
        std::reference_wrapper<T>,
        ShapeInternal>::type;

    ShapeHolderRef(std::reference_wrapper<T> x) : base_t{std::move(x)} { }

    ShapeHolderRef(adobe::move_from<ShapeHolderRef> x)
        : base_t{adobe::move_from<base_t>{x.source}} { }

    virtual double get_area_internal() const override
    { return this->get().get().get_area(); }

    virtual double get_perimeter_internal() const override
    { return this->get().get().get_perimeter(); }
};
```

We could now write a new `ErasedShapeRef` wrapper; however, to reduce duplication, it makes sense to turn the existing `ErasedShape` class into a class *template*, parameterized by the holder type:

```
template<template<typename> class HOLDER>
struct ErasedShape : adobe::poly_base<ShapeInternal, HOLDER>
{
    using base_t = adobe::poly_base<ShapeInternal, HOLDER>;

    using base_t::base_t; // INHERIT BASE CLASS'S CONSTRUCTORS

    ErasedShape(adobe::move_from<ErasedShape> x)
    : base_t(adobe::move_from<base_t>(x.source)) { }

    // SAME AS BEFORE...
};
```

This allows defining the final type-erasing wrappers as follows:

```
using Shape = adobe::poly<ErasedShape<ShapeHolder>>; // FOR VALUE SEMANTICS

using ShapeRef = adobe::poly<ErasedShape<ShapeHolderRef>>; // FOR REF. SEM.
```

This allows writing code like the following:

```
int main()
{
    Rectangle r{{1.0, 2.0}, 5.0, 6.0};

    ShapeRef s{std::ref(r)};

    std::cout << s.get_area() << std::endl; // PRINTS 30

    r.set_size(4, 2); // SUPPOSE SUCH A FUNCTION EXISTS

    std::cout << s.get_area() << std::endl; // PRINTS 8
}
```

Due to how the `adobe::poly` base class works, the call to `std::ref()` above cannot be omitted – although simple modifications to the library's source code would allow doing that.

A worse problem is that if a certain type U needs dedicated adaptation logic – like the `Triangle` class from Section 2.2.1 – one additional explicit specialization of `ShapeHolderRef` for `std::reference_wrapper<U>` needs to be provided, which complicates maintenance.

Another issue is the lack of support for rvalue references: `std::reference_wrapper` is meant to simulate lvalue references only, and its constructor accepting an lvalue reference is deleted. Hence, creating a new eraser `ShapeRefRef` would be problematic and may require changes to the library's source code.

Finally, Adobe.Poly suffers from a problem which is common to all library solutions for DGP, at least concerning reference semantics: the Proxy Dilemma [2]. The problem stems from the fact that a referencing type-erasure wrapper is itself a *distinct object* from the object it erases. In other words:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

auto s = ShapeRef{std::ref(r)};

assert(&r == &s); // THIS ASSERTION ALWAYS FAILS!
```

This is an issue for compile-time generic algorithms written in the form of function templates: depending on how they are written, these algorithms may not be allowed to work transparently with objects accessed through a type-erasing wrapper. This situation is also described in [19] as part of the documentation of the Boost.TypeErasure library [5].

A related problem becomes apparent when trying to use smart pointers to type-erasing wrappers. For symmetry, one would expect the following initializations to be valid:

```
Shape s = Rectangle{{1.0, 2.0}, 5.0, 6.0};

std::unique_ptr<Shape> s = std::make_unique<Rectangle>({1.0, 2.0}, 5.0, 6.0);
```

Unfortunately, the second initialization is illegal, because the `unique_ptr` would internally hold a pointer to a `Shape`, which is not compatible with a pointer to a `Rectangle` (at a language level, those two types are unrelated). This can be worked around by creating a `Shape` object instead of a `Rectangle` object, but doing so will not help when we already have a `unique_ptr<Rectangle>` to begin with, and we want to pass it to a generic algorithm expecting a `unique_ptr<Shape>`.

2.2.3. Conclusions on Adobe.Poly

Adobe.Poly enables DGP for types with value semantics. While elegant and efficient – thanks to techniques such as SBO – it is not very easy to master, and requires writing a significant amount of boilerplate code, as shown in the previous sections.

This boilerplate code may be non-trivial due to the presence of templates and (possibly) template specializations; furthermore, it tends to become more complex when erasing overloaded operators such as the equality-comparison operator. For example, this is how our `ShapeHolder` would dispatch equality checks to the underlying object:

```
virtual bool equals(ShapeInternal const& rhs) const override
{ return this->type_info() == rhs.type_info() &&
         this->get() == *static_cast<const T*>(rhs.cast()); }
```

Support for reference semantics with wrappers such as `std::reference_wrapper`, while not provided off-the-shelf, can be by obtained at the cost of writing some more boilerplate code; the amount of work starts to become a serious maintenance burden when adaptation is used instead of duck-typing.

Supporting reference semantics for rvalue references may not be possible without changing the library's code, and more inherent problems like the Proxy Dilemma [2] cannot be solved at all, limiting the possibility of mixing DGP with classical, compile-time GP.

Concerning build times, Adobe.Poly is a relatively light-weight library, because its internals are not heavily based on template meta-programming (TMP) techniques; yet, a certain overhead with respect to inheritance-based polymorphism is unavoidable.

As a final remark, although the documentation on the project's website does cover the details of the API to some degree, no introductory example or tutorial is available. This makes it very hard for a neophyte to understand how the library works, and how to make proper use of it.

2.3. Boost.TypeErasure

Boost.TypeErasure [5], authored by Steven Watanabe, is a recent addition to the Boost [11] suite of libraries, which first introduced it in version 1.54.

Thanks to a heavy use of TMP techniques and C-style macros, Boost.TypeErasure helps reducing the amount of boilerplate code users have to write in order to define their own type-erasing wrappers. However, this also means that compilation times are slower than those obtained with Adobe.Poly, and significantly slower than those obtained by using classical, inheritance-based polymorphism.

Another major disadvantage brought by TMP techniques is the quality of compiler diagnostics. Forgetting to specify a `const` qualifier where the framework expects it may result in pages of unintelligible error messages.

It is also worth mentioning that the amount of work needed to define custom concepts increases significantly when the advanced features of the library are used. One such situation is when adaptation is desired rather than duck-typing (see Section 2.3.1); another one is when it is necessary to type-erase operations that are not supported by the macro syntax (e.g. conversion operators [7]) . Finally, macros can only be defined at namespace scope, which may be a limiting factor at least for readability.

When the usage scenario is simple enough, on the other hand, Boost.TypeErasure does offer a very convenient notation. For instance, the following is all that is required in order to support the working example defined in Section 2.1 (compare this with the amount of code required by Adobe.Poly):

```
namespace erasure = boost::type_erasure;

BOOST_TYPE_ERASURE_MEMBER((has_get_area), get_area)
BOOST_TYPE_ERASURE_MEMBER((has_get_perimeter), get_perimeter)

using ShapeRequirements = boost::mpl::vector<
    erasure::copy_constructible<>,
    has_get_area<double(), erasure::_self const>,
    has_get_perimeter<double(), erasure::_self const>,
    erasure::relaxed>;

using Shape = erasure::any<ShapeRequirements>;
```

The `BOOST_TYPE_ERASURE_MEMBER` macro allows generating template code for concepts that require the presence of a member with a certain name in modeling types (e.g. `has_get_area` requires the presence of a member function called `get_area()`). For member functions, the signature is *not* specified as an argument to the `BOOST_TYPE_ERASURE_MEMBER` macro; rather, it is specified as the first template argument to the generated concept (e.g. `has_get_area<>`) when used inside a larger set of requirements (e.g. the `ShapeRequirements` compile-time sequence). The second template argument to the generated concept specifies whether the corresponding member function is `const`-qualified, in which case `_self const` should be used. When this argument is omitted, it defaults to `_self`, which denotes a mutating function.

The `copy_constructible` concept is a predefined concept provided by the library, and the `relaxed` constraint instructs the framework to use certain default settings that regulate the behavior of the library under particular circumstances – a deeper analysis is beyond the scope of this document.

The usage of compile-time MPL sequences of micro-constraints to define concepts, as well as the adoption of placeholders such as `_self`, do not help readability; however, a comparison with the amount of code required for the same use case by Adobe.Poly (see Section 2.2) makes the benefits of using Boost.TypeErasure apparent.

2.3.1. Adaptation vs. duck-typing

Let's suppose we want our `Shape` wrapper to be able to erase the class `Triangle` from Section 2.2.1. Because the names of the member functions of `Shape` do not match those of the member functions of `Triangle`, we cannot use the `BOOST_TYPE_ERASURE_MEMBER` macro as we did in the previous section. We therefore need to explicitly define one class template for each required member function:

```
template<class T>
struct has_get_area
{
    static double apply(T const& x) { return x.get_area(); }
};

template<class T>
struct has_get_perimeter
{
    static double apply(T const& x) { return x.get_perimeter(); }
};
```

Next we'll have to inform the framework about the existence of our concepts by specializing the `concept_interface` template (an explanation of the details is beyond the scope of this document):

```
namespace boost { namespace type_erasure
{
    template<typename C, typename Base>
    struct concept_interface<has_get_area<C>, Base, C> : Base
    {
        double get_area() const
        { return call(has_get_area<C>(), *this); }
    };

    template<typename C, typename Base>
    struct concept_interface<has_get_perimeter<C>, Base, C> : Base
    {
        double get_perimeter() const
        { return call(has_get_perimeter<C>(), *this); }
    };
} }
```

We will also have to rewrite our `ShapeRequirements` class, because the definition of the `has_get_area<>` and `has_get_perimeter<>` templates has changed, and the signature parameter is no longer required:

```
using ShapeRequirements = boost::mpl::vector<
    boost::type_erasure::copy_constructible<>,
    has_get_area<boost::type_erasure::_self>,
    has_get_perimeter<boost::type_erasure::_self>,
    boost::type_erasure::relaxed>;
```

This code is completely equivalent to the one we wrote in the previous section, only more verbose. However, it gives us enough flexibility to define custom adaptation logic by specializing the `has_get_area<>` and `has_get_perimeter<>` templates for the `Triangle` class:

```
template<>
struct has_get_area<Triangle>
{
    static double apply(Triangle const& x) { return x.compute_area(); }
};

template<>
struct has_get_perimeter<Triangle>
{
    static double apply(Triangle const& x) { return x.compute_perimeter(); }
};
```

This completes the adaptation work. Both readability and verbosity tend to match those of Adobe.Poly (see Section 2.2.1 for a comparison).

It is worth mentioning that things would become slightly more cumbersome if we wanted our concept templates to accept further template parameters, such as as the type of a function argument. The following example, taken from [20], shows how to specialize `concept_interface` for a generic constraint `has_push_back<T>` that requires the presence of a member function `push_back()` accepting an argument of type `T`:

```
namespace boost { namespace type_erasure
{
    template<class C, class T, class Base>
    struct concept_interface<has_push_back<C, T>, Base, C> : Base
    {
        void push_back(typename as_param<Base, T const&>::type arg)
        { call(has_push_back<C, T>(), *this, arg); }
    };
} }
```

2.3.2. Reference semantics

Unlike Adobe.Poly, Boost.TypeErasure offers built-in support for reference semantics, and taking advantage of it is very simple. The semantics of an erasing wrapper is specified when instantiating the `any` class template. In Section 2.3 above we defined our `Shape` wrapper as follows:

```
using Shape = erasure::any<ShapeRequirements>;
```

In order to turn `Shape` into a wrapper with reference semantics, it is sufficient to provide an additional template argument to `any`, like so:

```
using ShapeRef = erasure::any<ShapeRequirements, erasure::_self&>;
```

This allows us to write code similar to the one from Section 2.2.2 (notice, that using `std::ref()` is not required here):

```
int main()
{
    Rectangle r{{1.0, 2.0}, 5.0, 6.0};

    ShapeRef s{r};

    std::cout << s.get_area() << std::endl; // PRINTS 30

    r.set_size(4, 2); // SUPPOSE SUCH A FUNCTION EXISTS

    std::cout << s.get_area() << std::endl; // PRINTS 8
}
```

Notably, rvalue references are supported in a similar way. In fact, it is possible to define an erasing wrapper like so:

```
using ShapeRefRef = erasure::any<ShapeRequirements, erasure::_self&&>;
```

The `ShapeRefRef` wrapper could then be initialized as shown below:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

auto s = Shape{r};

ShapeRefRef s1 = std::move(r); // OK

ShapeRefRef s2 = std::move(s); // OK
```

Unfortunately, like every library solution, Boost.TypeErasure is unable to solve the Proxy Dilemma [2], and therefore suffers from limitations similar to those described in Section 2.2.2. In particular, compile-time generic algorithms are not guaranteed to work transparently with erasing wrappers [19], and ownership of type-erased objects through smart pointers is limited in flexibility.

2.3.3. Conclusions on Boost.TypeErasure

Boost.TypeErasure is a powerful, modern C++ library that supports DGP with both value semantics and reference semantics. It makes simple usage scenarios relatively easy to handle by providing macros that improve readability and predefined concepts that save the user some repetitive work.

However, the syntactic convenience Boost.TypeErasure with respect to Adobe.Poly tends to fade away when using adaptation rather than duck-typing; more generally, accomplishing non-trivial tasks through Boost.TypeErasure requires (1) being acquainted with TMP techniques and (2) being ready to interpret long, hard-to-decipher compiler errors – this may sound less problematic then it actually is.

For a demonstration of the kind of complexity involved with solving relatively simple problems, the reader is invited to refer to the documentation page showing how to define a concept which requires the presence of two different overloads of a member function [21].

Complexity aside, Boost.TypeErasure offers a lot of interesting functionality, such as support for associated types, covariant function arguments, and implicit type-safe conversions between concepts. It also has a macro `BOOST_TYPE_ERASURE_FREE` which allows defining freestanding function requirements pretty much in the same way as the `BOOST_TYPE_ERASURE_MEMBER` macro allows defining member function requirements (see Section 2.3).

The limits of Boost.TypeErasure are those inherent to all library solutions for DGP: reference semantics through smart pointers is not easy to achieve, and the Proxy Dilemma [2] cannot be solved.

Last but not least, the expressivity and flexibility of Boost.TypeErasure come at the expense of relatively high build times, due to the heavy use of TMP techniques in the library's internals.

2.4. Pyry Jahkola's Poly

The Poly Library [9], authored by Pyry Jahkola, is a modern C++ library inspired by Adobe.Poly. It is not meant for production use and lacks a few fundamental features (like support for reference semantics), so it will not be covered extensively in this document. For documentation, the reader can refer to [9].

By design, Poly does not support member function call syntax on type-erasing wrappers: instead, it forces the use of freestanding functions. Therefore, our function `print_areas()` from Section 2.1 will have to be rewritten like so:

```
void print_areas(std::vector<Shape> const& v)
{
    for (auto const& s : v)
    {
        std::cout << "Area: " << get_area(s) << std::endl;
    }
}
```

The first step consists in defining the interface of our type-erased wrapper. This is easily done by (1) using the `POLY_CALLABLE()` macro to generate callable objects with the names of the functions required by our `Shape` concept, and (2) instantiating the `poly::interface<>` variadic class template with a corresponding list of function signatures:

```
#include <poly/interface.hpp>

POLY_CALLABLE(get_area);
POLY_CALLABLE(get_perimeter);

using Shape = poly::interface<
    double(get_area_, poly::self const&),
    double(get_perimeter_, poly::self const&)>;
```

The first argument type of each concept function signature should be the name of the corresponding callable defined through the `POLY_CALLABLE()` macro, with an underscore appended.

Then, we shall define one function template named `call()` – which will be recognized by the framework – for each function required by our concept; this implements the dispatch mechanism:

```
template<typename T>
double call(get_area_, T const& x)
{
    return x.get_area();
}

template<typename T>
double call(get_perimeter_, T const& x)
{
    return x.get_perimeter();
}
```

This is everything that needs to be in place for realizing the use case from Section 2.1. When adaptation is necessary, e.g for type-erasing the `Triangle` class from Section 2.2.1, proper (non-template) overloads of the `call()` function templates must be defined to specialize the dispatch:

```
double call(get_area_, Triangle const& x)
{
    return x.compute_area();
}

double call(get_perimeter_, Triangle const& x)
{
    return x.compute_perimeter();
}
```

The way Poly enables DGP in C++ is elegant and simple, but the library lacks a few important features. In particular, as already mentioned, it does not support reference semantics by design – and therefore does not attempt to address the Proxy Dilemma [2]; the design choice of forcing the use of freestanding functions on type-erasing interfaces also limits its generality to some extent.

2.5. Boost.Interfaces

Jonathan Turkanis's Boost.Interfaces library [10] (which, in spite of its name, is not related to the Boost C++ Libraries) is part of Christopher Diggins's Object-Oriented Template Library (OOTL) library [22]. Boost.Interfaces is one of the first open-source projects that historically tackled the problem of supporting DGP in C++. Unfortunately, both Boost.Interfaces and the OOTL library seem to be no longer maintained, and their obsolete dependencies made it impossible for us to evaluate them. Besides, Boost.Interfaces apparently never evolved from an experimental stage into a real-world product ready for production use.

Based on the documentation available on the project's website, Boost.Interfaces allows creating custom type-erasing wrappers by specifying the interface that erased types have to adhere to by using macros. Whether a type conforms to an erasing wrapper's interface is determined exclusively through duck-typing: Boost.Interfaces does not support adaptation.

Boost.Interfaces's strong point is simplicity. For instance, this is how the type-erasing `Shape` wrapper for our working example from Section 2.1 would be defined:

```
BOOST_IDL_BEGIN(Shape)
    BOOST_IDL_FN0(get_area, double)
    BOOST_IDL_FN0(get_perimeter, double)
BOOST_IDL_END(Shape)
```

There are several downsides to this approach: in particular, the requirements for movability and copyability cannot be expressed through this simple mechanism. Perhaps this is one of the reasons why Boost.Interfaces only supports reference semantics.

To allow unique and shared ownership of type-erased objects, Boost.Interfaces provides its own smart pointers, which are meant to emulate the behavior of standard smart pointers – for more information, see [23].

2.6. Signatures

Unlike all the projects mentioned so far, Signatures [12] attempts to introduce support for DGP in C++ at a core language level. An implementation of Signatures as a language extension was available in GCC up to version 2.95 [13] [14].

The key notion introduced by Signatures in the C++ language is – unsurprisingly – that of a *signature*. A signature defines an interface that modeling types have to conform to in order to be treated polymorphically through a signature pointer or signature reference. The mechanism is non-intrusive: modeling types are not required (in fact, they are not allowed) to derive from a signature.

Like Boost.Interfaces (see Section 2.5), Signatures only focuses on reference semantics, and does not allow using type-erased wrappers with value semantics. Therefore, the original working example from Section 2.1 will have to be rewritten.

This is how the reworked `print_areas()` function would look like:

```
void print_areas(std::vector<Shape*> const& v) // VECTOR OF POINTERS!
{
    for (auto const s : v)
    {
        std::cout << s->get_area() << " ";
    }
}
```

The `main()` function also needs to be modified accordingly:

```
int main()
{
    auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

    auto c = Circle{{0.0, 0.0}, 42.0};

    std::vector<Shape*> v{&r, &c}; // VECTOR OF POINTERS!

    print_areas(v);
}
```

Finally, the type-erasing wrapper `Shape` would be defined like so:

```
signature Shape
{
    double get_area();
    double get_perimeter();
};
```

Signature functions cannot be `const`-qualified, yet they are capable of delegating to types that have `const`-qualified versions of those functions – this is something we will get back to in Sections 4 and 5.

A quick comparison with the previous sections reveals the simplicity and elegance of this approach: four lines of readable code involving no template meta-programming and no macros is everything which is needed to realize our use case, at least when duck-typing is sufficient.

Signatures offers support for adaptation through so-called *views*, but the syntax is largely sub-optimal. In particular, the adaptation logic for a given type T to a given signature S is not defined once and reused wherever necessary; instead, it must be specified every time an expression of type T (resp. T*) is used to initialize a variable of type S& (resp. S*):

```
Triangle t{{0.0, 0.0}, {1.0, 0.0}, {0.0, 1.0}};

S* p = (S*; get_area = compute_area, get_perimeter = compute_perimeter)&t;
```

Perhaps the most interesting aspect of Signatures is its capability of solving the Proxy Dilemma [2]: thanks to special compiler support, generic algorithms are guaranteed to work transparently with (pointers and references to) signatures just like they do with (pointers and references to) objects that adhere to those signatures. Moreover, since signature pointers are treated just like any other pointer type at a language level, ownership through smart pointers would be possible without introducing dedicated wrappers and without modifying the existing smart pointer types.

The implementation technique adopted by Signatures [14] is the same as the one proposed for implementing virtual concepts in Section 12 of this document – at least for what concerns reference semantics.

2.7. Conclusions

In this section we have analyzed five existing solutions for DGP in C++. Two of them, namely Adobe.Poly [8] and Boost.TypeErasure [5], are robust, well-known open-source libraries ready to be used in a production environment. The former is mostly focused on supporting value semantics and requires a certain amount of boilerplate code to be written by the user; the latter supports both value semantics and reference semantics, and tends to require less work for simple use cases. However, it heavily relies on TMP techniques, which slows down build time, and tends to become unwieldy for more complex usage scenarios. Both Adobe.Poly and Boost.TypeErasure support duck-typing as well as adaptation.

We also discussed Pyry Jahkola's Poly library [9], which is not ready for production use at the moment of writing, but provides a very convenient syntax for generating type-erasing wrappers. By design it only allows using freestanding functions on concept interfaces, and does not support reference semantics.

The Boost.Interfaces project [10] (part of the OOTL library [22]) is probably the earliest attempt to address the problem of DGP in C++ at a library level, and seems to be no longer under active development. Boost.Interfaces is only concerned with reference semantics, and allows defining type-erasing wrappers by using preprocessor macros. Adaptation is not supported.

None of the library solutions mentioned so far are capable of solving the Proxy Dilemma [2], which limits the possibility of using transparently type-erasing wrappers and the types they erase in generic algorithms; another consequence of creating wrapper proxies is that existing smart pointers are not suitable for owning type-erased objects, and dedicated smart pointers need to be provided instead; finally, supporting proper overload resolution in the presence of type-erasing wrappers is likely to require complicated user-defined conversions.

The Signatures project [12] tries to address the problem at a core language level by introducing a new language construct (*signatures*) and dedicated compiler support. Signatures was implemented as a language extension in GCC up to version 2.95 [13]. It only supports reference semantics, but does solve the Proxy Dilemma and therefore is not affected by the issues mentioned in the previous paragraph.

3. Design goals

This section states the fundamental goals that our approach to DGP pursuits and describes their impact on the design of the solution we propose. In approximate order of importance, our goals are:

- Simplicity and usability
- Support for value semantics and reference semantics
- No proxy objects
- Efficiency
- Integration with related language and library features
- Fast build times

The following sections will expand on each of these points.

3.1. Simplicity and usability

Type-erasure is, in principle, a simple idea. Existing library solutions require the user to write too much boilerplate code, especially for realizing non-trivial use cases. Even when the amount of work needed is not considered a limiting factor, the result tends to be code which is hard to read and/or maintain. Maintenance is especially problematic due to the very low quality of compiler diagnostics, which can easily consist of pages of template instantiation stack traces.

Endorsing Bjarne Stroustrup's mission of "*Making Simple Tasks Simple*" [24], our goal is to provide users with tools that (1) do not require them to deal with TMP unless parameterization is part of their *domain* problem, (2) do not require them to use macros in order to improve readability, and (3) do not require them to write boilerplate code and/or perform repetitive tasks in order to realize their use cases.

In our opinion, language support is a *condicio sine qua non* to achieve these goals, and the Signatures project [12] shows the potentials of a language-level solution for DGP in terms of simplicity and flexibility – to get an idea, compare the code in Section 2.6 with the snippets from Sections 2.2-2.5.

3.2. Support for value semantics and reference semantics

Value semantics is a popular programming style in the C++ community [25], and the C++ language encourages the definition of types that "behave like an `int`". The benefits of value semantics and the need for a non-intrusive form of polymorphism that supports them are well explained in [16], [18], and [26]. Supporting value semantics with DGP is therefore a crucial aspect of our solution.

However, it is sometimes necessary or convenient to adopt reference semantics for solving certain problems. The C++ language supports reference semantics through polymorphism based on inheritance and pointers/references; unfortunately, inheritance is an intrusive mechanism that causes the series of problems discussed in Section 1.

It is therefore important to provide support for DGP with reference semantics as well, and to do so in a way that integrates seamlessly with all the elements of modern C++ programming, including smart pointers and rvalue references.

3.3. No proxy objects

One of the problems that affect library solutions supporting reference semantics is the so-called Proxy Dilemma [2]. In order to access objects through an interface different from the one naturally defined by their types, pointer-like proxy objects must be created that perform the necessary adaptation.

These proxy objects are separate objects from the ones they are meant to abstract. In other words, supposing that Shape is some sort of type-erasing wrapper akin to the ones introduced in Section 2, the assertion in the code below will always fire:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

auto s = ShapeRef{r};

assert(&r == &s); // THIS ASSERTION ALWAYS FAILS!
```

This has a number of implications: firstly, generic algorithms might not work transparently with wrapper types as they do with the types erased by those wrappers [19]; secondly, different kinds of wrappers are necessary to support the different kind of references that exist in C++ (namely, *lvalue* and *rvalue* references); thirdly, reference-like wrappers cannot be used just like C++ references, since the language forbids overloading the dot operator for accessing class members; finally, ownership of type-erased objects with dynamic storage duration through smart pointers would require introducing dedicated smart pointer classes or specializing existing ones. All of these issues introduce unnecessary complexity (see Section 3.1) and/or limit expressivity (see Section 3.2).

We therefore require the following to be valid C++ code, and the assertion in the last line to always succeed:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

Shape& s = r;

assert(&r == &s); // THIS ASSERTION SHOULD ALWAYS SUCCEED
```

3.4. Efficiency

Efficiency is a primary concern for C++ developers as well as for the designers of the language. Our proposed solution aims at supporting DGP with no performance penalty with respect to the idiomatic solutions currently in use while guaranteeing a performance improvement in most situations.

Concerning reference semantics, the solution we propose tends to be superior to inheritance-based polymorphism because types are not required to be polymorphic (i.e. to have at least one virtual function) in order to be accessed through type-erasing virtual concepts; more to the point, this means that the instances of those types do not need to carry a *vtable* pointer (or several such pointers) regardless of whether they will be used polymorphically or not. This minimizes memory consumption.[1]

When objects are accessed polymorphically through virtual concepts, the dispatch mechanism is no different from the one that realizes a regular virtual function call; this means that our solution is performance-wise no worse than inheritance-based polymorphism.

Using heterogeneous types with value semantics polymorphically is not allowed at a language level, so the term of comparison for our proposal in this respect are the library solutions described in Sections 2.2-2.4.

The main technical obstacle with supporting value semantics is that, in general, the size of an erased object is unknown at compile-time. The consequence is that although an object which is used with value semantics has automatic storage duration (because its lifetime ends when execution leaves the block it is declared in), it cannot be allocated on the stack. Instead, the storage which holds the object must be allocated from the free store.

Dynamic memory allocation is an expensive procedure which might be problematic or unaffordable in certain scenarios. The performance penalty can be alleviated by adopting optimization techniques such as SBO and/or by providing custom allocation mechanisms. While SBO is covered in Section 12, the current version of this proposal is not concerned with customizing the allocation process. Existing library solutions do not offer this flexibility either.

In any case, the performance implications of dynamic allocations will only be relevant when using types with value semantics polymorphically: other use cases (e.g. polymorphic access of objects with reference semantics) are not affected. This is in line with the well-known motto "*you don't pay for what you don't use*", which is also one of C++'s fundamental design principles.

[1] The most natural implementation of concept pointers and concept references is as a pair of regular pointers (see Section 12 for the details). In situations where the number of pointers or references largely exceeds the number of objects, this solution will lead to higher memory consumption. Moreover, if concept pointers are copied around frequently and/or in performance-critical portions of an application, a penalty in terms of speed may also be observed. However, invoking functions through a concept pointer might be a more cache-friendly operation than invoking functions through a regular virtual table, because it would not be necessary to fetch the object from main memory in order to access its *vtpr*.

3.5. Integration with related language and library features

Introducing language support for DGP obviously calls for dedicated syntax in which to express the interface of type-erasing wrappers. One of the aims of this proposal is to integrate seamlessly with related language features rather than clashing with them on the syntactic (and semantic, where possible) plane.

Hence, our proposal allows reusing concepts as defined by the Concepts TS [6] with only a few limitations. In particular, the `Shape` wrapper from our working example in Section 2.1 would be automatically generated by the compiler from the following concept definition:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
};
```

In terms of clarity this is close to the syntax offered by Signatures [12], but more expressive – as we will see in Section 4 and 5. The dynamic usage of a concept like `Shape` would be disambiguated from its static usage through the `virtual` keyword. For instance:

```
void print_areas(std::vector<virtual Shape*> const& v) // NOT A TEMPLATE!
{
    for (auto const& s : v)
    {
        std::cout << s->get_area() << " ";
    }
}
```

An alternative way of writing the above would be to qualify the concept as `virtual` in its definition. This would make it unnecessary to explicitly qualify its usages:

```
template<typename T>
virtual concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
};

void print_areas(std::vector<Shape*> const& v) // NOT A TEMPLATE!
{
    for (auto const& s : v)
    {
        std::cout << s->get_area() << " ";
    }
}
```

Similar considerations, which have been partly covered in Section 3.3, can be made for the integration of virtual concepts with C++'s type system. In particular, forming concept pointers and references should be possible with the same syntax and semantics used for regular compound types.

Given the last definition of Shape given above, for example, the following initializations should all be legal and have the obvious meaning, and no assertions should be triggered:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

Shape& s = r;

Shape const* p = &r;

std::unique_ptr<Shape> up = std::make_unique<Rectangle>(r);

assert(up.get() == &r); // SHALL NEVER FIRE

std::shared_ptr<Shape> sp = std::move(up);

assert(sp.get() == &s); // SHALL NEVER FIRE

assert(&r == &s); // SHALL NEVER FIRE
```

Extensions to existing language features are sometimes needed to cover all the relevant use cases for type erasure, e.g. adaptation; while C++0x's *concept maps* [27] (i.e. adaptation in the context of static polymorphism) were dropped from the scope of the Concepts TS [6], adaptation is a common use case for type erasure, which is why we chose to propose a syntax to support it.

However, since we also chose to establish a sort of isomorphism between virtual concepts and compile-time concepts as defined in Concepts TS, this raises the question whether it would not be wise for this feature to be developed in parallel on both shores of the river – i.e. for DGP and classical GP with templates. We chose to provide some syntax to express adaptation, while remaining open to the possibility of dropping this feature for the sake of consistency and better integration with Concepts TS if deemed necessary.

Interaction with other language features such as inheritance, templates, and overload resolution ought to be carefully considered and will be discussed in detail throughout this document (in particular, see Sections 9 and 10).

3.6. Fast build times

One of the greatest benefits of inheritance-based polymorphism is that it enables a form of *dependency inversion* [3] thanks to which a software component can transfer execution to another software component at run-time without depending on its definition at compile-time.

Thanks to the layer of indirection provided by virtual functions and inheritance, modifying the definition of any type which implements an abstract interface does not require recompiling any of the clients using that interface. Apart from enabling independent deployment, inheritance and virtual functions represent a compilation firewall that helps keeping build times manageable in spite of C++'s sub-optimal compilation model.

One of our concerns with existing library solutions for DGP in C++ is that they tend to rely heavily on TMP techniques in order to save the user from dealing with repetitive tasks; this, unfortunately, leads to a dramatic increase in build times, making the benefits of a compilation firewall less perceivable.

One of the goals of our solution is therefore to support DGP with a compilation overhead comparable to the one implied by traditional, inheritance-based polymorphism; in particular, virtual concepts should lead to build times significantly faster than those obtained with the library solutions examined in Sections 2.2-2.5.

4. Virtual concepts in a nutshell

4.1. Non-intrusive virtuality

Mention again that concepts can be either defined as virtual or virtualized “on-demand”, if possible. Mention that there are a few restrictions (no disjunction of constraints), and refer to the next section for the details.

Syntax of templates, different semantics.

Also, virtual concepts can be “unvirtualized” on-demand using “static”.

Repeated occurrences of same concept name in a function signature.

Template syntax for non-template functions (symmetry with non-template syntax for template functions).

4.2. Extreme erasure: run-time `auto` and `std::any`

```
template<typename T>
concept bool Void = true;
```

For orthogonality, also `virtual auto` (this can only be taken by reference). What can you do with it? Nothing, except from casting it:

```
void foo(virtual auto& x)
{
    auto i = dynamic_cast<int>(x); // THROWS std::bad_cast ON FAILURE

    // ...
}
```

Any type models Void (virtual auto).

Difference between static_cast and dynamic_cast in case of concepts is the same as `static_cast<>` and `dynamic_cast<>` in case of other types (UB the first, exception the second). At run-time, that the iptr can easily get type information.

Also important is that this cast is generic, not specific for *one* specific type-erasure wrapper (e.g. std::any has `any_cast<>`, and does not distinguish between static and dynamic).