Instantiation context aware types in C++

Zsolt Parragi, Zoltán Porkoláb

Abstract: Programming languages usually limit the information available to types. Object oriented languages, such as C++ or Java typically define types as definition context aware, but instantiation context free: types are able to use other types and language constructs declared or defined before their definition, but they aren't aware of where, or in which context they are instantiated. While this is practical in most situations, both for considering the compilation speed and the complexity of the language, it also limits the expressiveness of the language. In this article, we present a way instanation context aware types could be implemented in modern C++, and a few examples where they could be useful.

Keywords: C++, Dependency Injection, Templates, Metaprogramming

Motivation

Types in most programming languages are aware of their definition context, but have no information about their instantiation context. This can be shown on a simple example:

```
// ...
struct T1 {
  // ...
};
struct T3;
struct T2 {
  T1 t1;
  T3* t3;
  // ...
};
// ...
struct Foo {
  T1 a;
  T2 b;
  T3 c;
};
```

In this simple program snippet, the types, such as T1 or Foo, are aware of the context of their definition. For example `Foo` is able to access the definition of `T1` as long as it is defined before it, or similarly, `T2` is able to use `T1`, which is defined before, or `T3*`, as `T3` is forward declared before its definition. Similarly it also would be able to use global variables, global functions, or member functions defined in existing types, if their access restrictions allows. This demonstrates that types in C++ are aware of their definition context.

Howewer, the language specification also states that `b`, an instantiation of `T2` in `Foo` has no knowledge about `a`, another data member defined just before it in `Foo`, or `c`, the data member defined after it, but still in the same scope. Similarly, `T1` does not know that it is instantiated in `T2` and `Foo`, or how many other data members, functions `Foo` has, or anything about their types. As for data members, the direct scope containing the type is the enclosing type, this means that types in C++ aren't aware of their instantiation context.

This limitation is intended, both for the sake of the compilers, and developers:

From a compilers perspective, it reduces the complexity of the implementation, and also introduces flexibility: It reduces the complexity, because the compiler does not have to do additional computations for each type for each instantiation – which, as instantiations are usually more numerous than definitions, would likely have a significant impact on compilation times. And while it is a limitation, by adding this restrictions, it also introduces flexibility: instantation context awareness is only possible, if the compiler evaluates the definition of everything – including member functions – in the type for the type for every instantiation. This is only possible if the compiler has the ability to generate a different intermediate or binary result each time, for which it needs access to the source each time. For this,

similarly to how templates work, the compiler would have to see the actual source code for everything related to that type – only not only for templates, but for every type. By making types instantation context free, the languange can be more flexible at other places – for example, it is able to allow "hidden" definitions for member function bodies.

From the developers perspective, added complexity isn't always better. As instantiation information is not required for most types, removing this restriction would only increase the possible errors in most programs, and would make runtime or compile time diagnostic harder.

Dependency Injection

Howewer developers sometimes do need this feature, or something similar, and compensate for its unavailability with design patterns. The most trivial example for this is when the data members of a type have some dependencies between each other. In the above example, `T2` holds a pointer to an instance of `T3`, which oftem injected as a parameter to its constructor, or as a setter. This principle, referred as dependency injection[5] is also supported by several frameworks, such as Boost DI[4].

These dependencies are also commonly represented as interfaces instead of concrete types, adding the ability to use different implementations as parameters. While this principle aims to help decoupling of the implementations, it is easy to misuse: when working with several implementations of some interfaces, sometimes the implementations rely on hidden properties of the type implementing the other interface instead of only on what is visible on the public interface. There could be many reasons for this, for example as a performance optimization, a design simplification, or simply forcing something new to work using legacy interfaces. Code relying this usually uses a `dynamic_cast`, or sometimes simply a `static_cast` on the interface in its implementation, and results in failures when invoking with an incorrect parameter.

Even when this isn't the case, an user of an interface could add additional requirements to the dependencies it accepts: for example on the surface it could accept a generic container, but a commend and a runtime check in the constructor would check that the given container only holds at most `N` items – otherwise the constructor would throw, preventing the creation of the object.

This is a neccessary side effect of using dependency injection: this technique allows the use of a runtime configuration when initializing a software product, theoretically starting up the program in several possible combinations. With this generalization, the developers either have to either check the validity of every possible combinations during the compilation of the program, or have to fall back on initialization time error messeges. As the latter is easier to do, and still results in an early error detection, it is the oblivious choice.

In practice however dependency injection is often used as an internal helper for development instead of a configuration option for end users: the dependencies are hard coded into the software, and the only goal of the DI framework is to produce a cleaner, better structured code, with less hand written boilerplate. In this case, the actually used configurations are known at compilation time, still, part of the error checks are deferred until the actual execution of the program. Even worse, these `if`s and `throw`s can be replaced with `assert`s, as developers assume that the code is throughly tested in a debug build, and want to increase the runtime performance of the released program.

This is where instantiation context aware types could help: if the required checks could be made during compilation time with the specific types known, developers could change runtime checks only present in debug builds into compile time checks, resulting in earlier and consistent problem detection.

Memory efficient dependencies

Additionally, if the types have enough information about their instatiation context, and the other types in that context, they could deduce the address of the parent type and other data members without storing actual pointers to them. This would result in a technique which not only allows additional compile time checks for the used types, but also presents a way for a memory-efficient dependency injection system, or rather, in the use of the composition pattern[1] as the dependency injection framework. While memory efficiency is not that important in perconal computers or handheld devices with the amount of memory available, in microcontrollers with only a few kilobytes of available memory it is still a primary question.

With context aware types, writing memory efficient, but also generic and safe types becomes a possobility, which could help embedded developers in writing better code. A possible use of this technique, which could be useful even on desktop systems is a memory efficient obvervable type: a type where every data member could be independently observed, without adding additional data members to every member type - and thus, increasing their size. Instead all data members could store their obververs in a common list, while maintaining the illusions that the user code directly attaches obsevrvers to the members.

```
struct StructWithObservableMembers {
  Observable<int> a;
  Observable<bool> b;
  Observable<floant> c;
private:
  ListOfObserversForEveryMember observers;
};
// ....
StructWithObservableMembers s;
s.a.observe([](...){ ... });
```

Conclusion

We have shown two examples where context aware types could result in definite advantates, with earlier validation, or with increasing the memory efficiency of the program. While this technique is not supported out of the box by traditional object oriented languages, the features provided by modern C++ standards allow a possible implementation with the use of templates. We developed a prototype framewrok using either the C++17 standard[2] with some commonly available compiler extensions, or with features currently present in the C++20 working draft[3]. While our prototype requires changes in the types which have to be instantation context aware, and also for the types that contain instantation context aware types, it also handles both of the examples mentioned previously, and could provide a desing alternative in some situations.

References

[1] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software* . Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. isbn : 0-201-63361-2.

[2] ISO *ISO/IEC 14882:2017 Information technology - Programming languages - C++.* In: Geneva, Switzerland: International Organization for Standardization, 2017.

[3] ISO *Working Draft, Standard for Programming Language C++.* 2018. url: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4727.pdf (visited on 03/31/2018)

[4] Krzysztof Jusiak. *[Boost].DI.* 2018. url: http://boost-experimental.github.io/di/ (visited on 03/31/2018)

[5] Dhanji R. Prasanna. *Dependency Injection*. 1st. Greenwich, CT, USA: Manning Publications Co., 2009. ISBN: 193398855X, 9781933988559