Invariant, covariant and contravariant

Freitag, 21. April 2023 10:18

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

These terms are related to polymorphism and inheritance.
Variant is especially interesting in terms of the generic programming: class templates and parameterized types relationship - subtyping.

Invariant

Generally, the relationship of the template parameters doesn't impose any relationship of the class template itself.
That is to say, if we have class template

```
template <class T>
class A{};
```

Having "is-a" relationship: **Base <= Derived** doesn't imply

```
A<Base> <= A<Derived>
```

that relationship is preserved.

In other words, that in every context where the *A<Base>* is expected, we can use the *A<Derived>*, since *A<Derived>* is superset of the *A<Base>*, in the same way as *Derived* is superset of the *Base* - that is, *A<Based>* is contained into *A<Derived>*, in the same way as the *Base* is contained into *Derived*.
This is known as covariant.

For example, all standard containers (std::vector<T, Allocator>) are invariant, for the very same reason as mentioned: relationship on template parameter T doesn't yield type substitution.

Covariant

For class template A

```
template <class T>
class A{};
```

if the relationship of parameterized type
Base <= Derived: does imply

```
A<Base> <= A<Derived>,
class A<Derived>: public A<Base> {};
```

we say that class template A is covariant on type T, i.e. it preserves "is-a" relationship of the template parameters.
@note: Private inheritance is a way to reuse the base class implementation (similar to composition), redefining - exposing functionality through the completely new contract: the "is-a" relationship is broken, and therefore the subtyping as well.
This is known as "has-a" relationship.

Covariant return types is related with the runtime polymorphism and ability to override virtual method base class on return type - returning the "different" type - subtype of the matching base class virtual method ([1]).
This works with raw pointers (references), but not with the smart pointers, because the smart pointers are not covariant (according to above definition).
There is a way around, to use them indirectly: to avoid having interface with naked pointers - with ducking type & virtual dispatching ([2])

```
class BaseFactory
{
    public:
        // Ducking
        std::unique_ptr<Base> create() const {
            return do_create();
        }
    private:
        // Virtual dispatching
        virtual std::unique_ptr<Base> do_create() const = 0;
};

class DerivedFactory :public BaseFactory
{
    public:
        std::unique_ptr<Derived> create() const {
            return std::make_unique<Derived>();
        }
```

```cpp
    private:
        virtual std::unique_ptr<Base> do_create() const override {
            return create();
        }
};
```

Contravariant

Similar, for the class template A

```cpp
template <class T>
class A{};
```

If the relationship of parameterized type
Base <= Derived: does imply

`A<Base> >= A<Derived>`, as if
```cpp
class A<Base>: public A<Derived> {};
```

we say that class template A is contravariant on T, i.e. it inverts the "is-a" relationship of the template parameters.
That means, in every context in which the *A<Derived>* is expected, we can provide its subtype *A<Base>*, since *A<Derived>* is now subset of the *A<Base>*.

This can be beneficial with **CRTP** (Curiously Recurring Template Pattern).
How?
We want to use the derived class implementation behind the base class interface, ensuring that polymorphically - the proper destructor of the derived class is being called, before the base class is destroyed.
For that to accomplish we can:
- Make the destructor of the base class virtual
    - This will imply the runtime polymorphism
    - Increase the size of the base class (and all derived classes) for the size of the virtual table pointer
- Introduce the proper **factory method** that
    - returns the **std::unique_ptr<Base, derived_deleter>**, since std::unique_ptr is not contravariant type, or
    - returns the **std::unique_ptr<Derived>** that can be stored into **contravariant std::shared_ptr<Base>**

 # CRTP - proper call of derived class destructor
https://godbolt.org/z/qnMT6c1aT

Links

1) Covariance and contravariance in C++ – Arthur O'Dwyer – Stuff mostly about C++ (quuxplusone.github.io)
2) Covariance with Smart Pointers - Simplify C++! (arne-mertz.de)
3) https://www.youtube.com/watch?v=Wp5iYQqHspg