# C++ Truths

A C++ blog on intermediate to advanced topics in C++ programming language features, standards, idioms, design patterns, and object-oriented programming in general.

Saturday, November 14, 2015

## Covariance and Contravariance in C++ Standard Library

Covariance and Contravariance are concepts that come up often as you go deeper into generic programming. While designing a language that supports parametric polymorphism (e.g., templates in C++, generics in Java, C#), the language designer has a choice between Invariance, Covariance, and Contravariance when dealing with generic types. C++'s choice is "invariance". Let's look at an example.

```
struct Vehicle {};
struct Car : Vehicle {};

std::vector<Vehicle *> vehicles;
std::vector<Car *> cars;

vehicles = cars; // Does not compile
```

The above program does not compile because C++ templates are invariant. Of course, each time a C++ template is instantiated, the compiler creates a brand new type that uniquely represents that instantiation. Any other type to the same template creates another unique type that has nothing to do with the earlier one. Any two unrelated user-defined types in C++ can't be assigned to each-other by default. You have to provide a copy-constructor or an assignment operator.

However, the fun starts when you realize that it's just a choice and there are other valid choices. In fact, C++ makes a different choice for pointers and references. For example, it's common knowledge that pointer of type Car is assignable to pointer of type Vehicle. That's because Car is a subtype of Vehicle. More accurately, the Car struct inherits from the Vehicle struct and the compiler allows us to use Car pointers in places where Vehicle pointer is expected. I.e., subtyping is activated through inheritance. Later in the post we will use subtyping without using inheritance.

If you think about pointers as a shortcut for the Pointer template below, it becomes apparent that the language has some special rules for them. Don't let the special * syntax confuse you. It is just a shortcut to avoid the ceremony below.

```
template <class T>
using Pointer = T *;

Pointer<Vehicle> vehicle;
Pointer<Car> car;

vehicle = car; // Works!
```

So what choices are available? The question we want to ask ourselves is, "What relationship do I expect between instantiations of a template with two different types that happen to have a subtype relationship?"

- The first choice is **no relationship**. I.e., the template instantiations completely ignore the relationship between parameter types. This is C++ default. It's called invariance. (a.k.a. C++ templates are invariant)
- The second choice is **covariant**. I.e., the template instantiations have the same subtype relationship as the parameter types. This is seen in C++ pointers and also in std::shared_ptr, std::unique_ptr because they want to behave as much like pointers as possible. You have write special code to enable that because the language does not give it to you by default.
- The third choice is **contravariance**. I.e., the template instantiations have the opposite subtype relationship to that of the parameter types. I.e., TEMPLATE<base> is subtype of TEMPLATE<derived>. We'll come back to contravariance in much more detail later in the post.

All C++ standard library containers are invariant (even if they contain pointers).

## Covariance

As said earlier, with covariance, the templated type maintains the relationship between argument types. I.e., if argument types are unrelated, the templated types shall be unrelated. If *derived* is a sub-type of *base* (expressed as inheritance) then TEMPLATE<derived> shall be sub-type of TEMPLATE<base>. I.e., any place where TEMPLATE<base> is expected, TEMPLATE<derived> can be substituted and everything will work just fine. The other way around is not allowed.

There are some common examples of covariance in C++ standard library.

```
std::shared_ptr<Vehicle> shptr_vehicle;
std::shared_ptr<Car> shptr_car;
shptr_vehicle = shptr_car; // Works
shptr_car = shptr_vehicle' // Does not work.

std::unique_ptr<Vehicle> unique_vehicle;
std::unique_ptr<Car> unique_car;
unique_vehicle = std::move(unique_car); // Works
unique_car = std::move(unique_vehicle); // Does not work
```

One (formal) way to think about covariance is that "the type is allowed to get **bigger** upon assignment". I.e., Vehicle is broader/bigger type than Car. Here's a quick rundown of some of the commonly used C++ standard library types and their covariance/contravariance properties.

| Type | Covariant | Contravariant |
|---|---|---|
| STL containers | No | No |
| std::initializer_list<T *> | No | No |
| std::future<T> | No | No |
| boost::optional<T> | No (see note below) | No |
| std::shared_ptr<T> | Yes | No |

| | | |
|---|---|---|
| std::unique_ptr<T> | Yes | No |
| std::pair<T *, U *> | Yes | No |
| std::tuple<T *, U *> | Yes | No |
| std::atomic<T *> | Yes | No |
| std::function<R *(T *)> | Yes (in return) | Yes (in arguments) |

The boost::optional<T> appears to be covariant but it really isn't because it slices the object underneath. The same thing happens with std::pair and std::tuple. Therefore, they behave covariantly correctly only when the parameter type itself behaves covariantly.

Finally, Combining one covariant type with another (e.g., std::shared_ptr<std::tuple<T *>>) does not necessarily preserve covariance because it is not built into the language. It is often implemented as a **single-level** direct convertibility. I.e., std::tuple<Car *> * is not directly convertible to std::tuple<Vehicle *> *. It would have been if the language itself enforced subtyping between std::tuple<Car*> and std::tuple<Vehicle *> but it does not. On the other hand, std::tuple<std::shared_ptr<T>> behaves covariantly.

By "single-level direct convertibility", I mean the following conversion of U* to T*. Convertibility is poor man's test for subtyping in C++.

A covariant SmartPointer might be implemented as follows.

```
template <class T>
class SmartPointer
{
public:
    template <typename U>
    SmartPointer(U* p) : p_(p) {}

    template <typename U>
    SmartPointer(const SmartPointer<U>& sp,
                 typename std::enable_if<std::is_convertible<U*, T*>::value, void>::
type * = 0)
      : p_(sp.p_) {}

    template <typename U>
    typename std::enable_if<std::is_convertible<U*, T*>::value, SmartPointer<T>&>::t
ype
    operator=(const SmartPointer<U> & sp)
    {
        p_ = sp.p_;
        return *this;
    }

   T* p_;
};
```

## Contravariance

Contravariance, as it turns out, is quite counter-intuitive and messes up with your brain. But it is a very valid choice when it comes to selecting how generic types behave. Before we deal with contravariance, lets quickly revisit a very old C++ feature: covariant return

types.

Consider the following class hierarchy.

```
class VehicleFactory {
  public:
    virtual Vehicle * create() const { return new Vehicle(); }
    virtual ~VehicleFactory() {}
};

class CarFactory : public VehicleFactory {
public:
    virtual Car * create() const override { return new Car(); }
};
```

Note that the return value of VehicleFactory::create function is Vehicle * where as CarFactory::create is Car *. This is allowed. The CarFactory::create function overrides its parent's virtual function. This feature is called overriding with covariant return types.

What happens when you change the raw pointers to std::shared_ptr? Is it still a valid program?....

As it turns out, it's not. std::shared_ptr (or any simulated covariant type for that matter) can't fool the compiler into believing that the two functions have covariant return types. The compiler rejects the code because as far as it knows, only the pointer types (and references too) have built-in covariance and nothing else.

Lets look a these two factories from the substitutability perspective. The client of VehicleFactory (which has no knowledge of CarFactory) can use VehicleFactory safely even if the create function gets dispatched to CarFactory at run-time. After all, the create function return something that can be treated like a vehicle. No concrete details about Car are necessary for the client to work correctly. That's just classic Object-oriented programming.

Covariance appears to work fine for return types of overridden functions. How about the argument? Is there some sort of variance possible? Does C++ support it? Does it make sense outside C++?

Let's change the create function to accept Iron * as raw material. Obviously, the CarFactory::create must also accept an argument of type Iron *. It is supposed to work and it does. That's old hat.

What if CarFactory is so advanced that it takes any Metal and creates a Car? Consider the following.

```
struct Vehicle {};
struct Car : Vehicle {};

struct Metal {};
struct Iron : Metal {};

class VehicleFactory {
  public:
```

```
    virtual Vehicle * create(Iron *) const { return new Vehicle(); }
    virtual ~VehicleFactory() {}
};

class CarFactory : public VehicleFactory {
public:
    virtual Car * create(Metal *) const override { return new Car(); }
};
```

The above program is illegal C++. The CarFactory::create does not override anything in its base class and therefore due to the override keyword compiler rejects the code. Without override, the program compiles but you are looking at two completely separate functions marked virtual but really they won't do what you expect.

More interesting question is whether it makes sense to override a function in a way that the argument in the derived function is broader/larger than that of the bases's?...

**Welcome to Contravariance...**

It totally does make sense and this language feature is called **contravariant argument types**. From the perspective of the client of VehicleFactory, the client needs to provide some Iron. The CarFactory not only accepts Iron but any Metal to make a Car. So the Client works just fine.

Note the reversed relationship in the argument types. The derived create function accepts the broader type because it must do at least as much as the base's function is able to do. This reverse relationship is the crux of contravariance.

C++ does not have built-in support for contravariant argument types. So that's how it ends for C++? Of course not!

## Covariant Return Types and Contravariant Argument Types in std::function

OK, the heading gives it away so lets get right down to an example.

```
template <class T>
using Sink = std::function<void (T *)>;

Sink<Vehicle> vehicle_sink = [](Vehicle *){ std::cout << "Got some vehicle\n"; };
Sink<Car> car_sink = vehicle_sink; // Works!
car_sink(new Car());

vehicle_sink = car_sink; // Fails to compile
```

Sink is a function type that accepts any pointer of type T and return nothing. car_sink is a function that accepts only cars and vehicle_sink is a function that accepts any vehicle. Intuitively, it makes sense that if the client needs a car_sink, a vehicle_sink will work just fine because it is more general. Therefore, substitutability works in the reverse direction of parameter types. As a result, Sink is contravariant in its argument type.

std::function is covariant in return type too.

```
std::function<Car * (Metal *)> car_factory =
  [](Metal *){ std::cout << "Got some Metal\n"; return new Car(); };
```

```
std::function<Vehicle * (Iron *)> vehicle_factory = car_factory;

Vehicle * some_vehicle = vehicle_factory(new Iron()); // Works
```

Covariance and Contravariance of std::function works with smart pointers too. I.e., std::function taking a shared_ptr of base type is convertible to std::function taking a shared_ptr of derived type.

```
std::cout << std::is_convertible<std::function<void (std::shared_ptr<Vehicle>)>,
                                 std::function<void (std::shared_ptr<Car>)>>::value
          << "\n"; // prints 1.
```

**Sink of a Sink is a Source!**

I hope the examples so far have helped build an intuition behind covariance and contravariance. So far it looks like types that appear in argument position should behave contravariantly and types that appear in return position, should behave covariantly. It's a good intuition only until it breaks!

```
template <class T>
using Source = std::function<void (Sink<T>)>;

Source<Car> source_car = [](Sink<Car> sink_car){ sink_car(new Car()); };

source_car([](Car *){ std::cout << "Got a Car!!\n"; });

Source<Vehicle> source_vehicle = source_car; // covariance!
```

Type T occurs at argument position in Source. So is Source contravariant in T?...

It's not! It's still covariant in T.

However, Source<T> is contravriant in Sink<T> though.... Afterall, Source is a **Sink of a Sink<T>**!

OK, still with me?

Let's get this *&%$# straight!

Source<Car> does not really take Car as an argument. It takes Sink<Car> as an argument. The only thing you can really do with it is sink/pass a car into it. Therefore, the lambda passes a new car pointer to sink_car. Again on the next line, calling source_car you have to pass a Sink<Car>. That of course is a lambda that accepts Car pointer as input and simply prints a happy message.

Source<Car> indeed works like a factory of Cars. It does not "return" it. It uses a callback to give you your new car. It's equivalent to returning a new Car. After all, the direction of dataflow is outward. From Callee to the Caller. As the data is flowing outwards, it's covariant.

More formally, type of Source is (T->())->(). A function that takes a callback as an input and returns nothing (i.e., read () as void). As T appears on the left hand side of even number of arrows, it's covariant with respect to the entire type. As simple as that!

## Generalizing with Multiple Arguments and Currying

The covariance and contravariance of std::function works seamlessly with multiple argument functions as well as when they are curried.

```
struct Metal {};
struct Iron : Metal {};
struct Copper : Metal {};

// multiple contravariant position arguments
std::function<Vehicle * (Iron *, Copper *)> vehicle_ic;
std::function<Car * (Metal *, Metal *)> car_mm = [](Metal *, Metal *) { return new C
ar(); };
vehicle_ic = car_mm;
vehicle_ic(new Iron(), new Copper());

// Curried versions
std::function<std::function<Vehicle * (Copper *)> (Iron *)> curried_vehicle;
std::function<std::function<Car * (Metal *)> (Metal *)> curried_car;
curried_car = [](Metal *m) {
  return std::function<Car * (Metal *)>([m](Metal *) { return new Car(); });
};
curried_vehicle = curried_car;
curried_vehicle(new Iron())(new Copper());
```

The car_mm function can be substituted where vehicle_ic is expected because it accepts wider types and returns narrower types (subtypes). The difference is that these are two argument functions. Each argument type must be at least the same as what's expected by the client or broader.

As every multi-argument function can be represented in curried form, we don't want to throw way our nice co-/contra-variant capabilities of the function-type while currying. Of course, it does not as can be seen from the next example.

The curried_vehicle function accepts a single argument and returns a std::function. curried_car is a subtype of curried_vehicle only if it accepts equal-or-broader type and returns equal-or-narrower type. Clearly, curried_car accepts Metal*, which is broader than Iron*. On the return side, it must return a function-type that is a subtype of the return type of curried_vehicle. Applying the rules of function subtyping again, we see that the returned function type is also a proper subtype. Hence currying is oblivious to co-/contra-variance of argument/return types.

So that's it for now on co-/contra-variance. CIAO until next time!

Live code tested on latest gcc, clang, and vs2015.

For comments see reddit/r/cpp and Hacker News.