Contracts, Types & Functions

Document #: P3583R0
Date: 2025-01-12
Project: Programming Language C++
Audience: SG21 Contracts Working Group
Reply-to: Jonas Persson
<jonas.persson@iar.com>

Contents

1 Introduction

Missing contracts for function pointers is a major hole in contracts proposal. This paper explore a solution to allow putting contracts in the type.

[P3327R0] discuss the solution space of contracts on function pointers, but is far to quick in dismissing putting them on the function type. It list a number of problem but don't go too deep into thinking about how they can be avoided.

[P2900R12] do not allow contracts on function pointers. Function with contracts implicitly converts to function pointers without contracts. This is motivated by backward compatibility with old code that should not break when contracts are added.

Even in the light of this, contracts in the types are desirable. It ensures contracts through pointer assignment chains. It has no extra runtime or size overhead. It can be used in templates.

This paper will try to explore what contracts on function types looks like and how they add the to the functionality of contracts and to the usability of functions in general.

2 Proposal

2.1 Overview

The major problems with contracts on types that we want to solve here is:

- Decide if the contracts are the same.
- How to do the name mangling.
- Avoid viral changes when improving code.

Function types today is a structural type. Functions defined in different places resolve to the same type if each of their subparts resolves to identical types. They have no names and a unique identifier must be created by name mangling.

When adding a contract to a type the compiler must be able to decide if two objects have the same type or not. To do this the expressions must be normalizable in a predictable way. Mangling the name has the same problem when trying to incorporate the contract predicate expressions into the name. It must be normalizable and it may also create exceptionally long names.

Both of these problems can be solved by letting function types with contracts not be structural types. Class types would have this very same problem if they where structural types, but they solve this by having a single definition with a name that other can refer to. The same can be done for function types. Function types without contracts are still considered structural types that are defined by its signature, but as soon a contract is added a nominal type is created.

Contract annotations on the type do not contribute to overload resolution. There are no implicit conversions between pointers of different types.

To support the [P2900R12] case with non-propagation, functions may still have implementation contracts that are not part of the type. This allows adding contracts incrementally without the viral property of types. The contracts described in [P2900R12] are then the implementation contracts.

This could be all of the proposal. But there is a hole in the current type system for functions in that there is no way to define functions from a predefined function type. This hole needs to be filled so that type safety can uphold when taking the address of a functions. Valid syntax need to be specified for free, member and lambda functions.

This will also open up new use cases outside contracts. There are many reasons it would be nice to create functions from function types. Sometimes there are several functions that needs to have the exact same signature. Other times it could be useful to be able to overload on callbacks with the same signature but different purpose. And a real problem today is the sheer size some type declarations have. Being able to separate out the type would enhance readability enormously.

2.2 Contracts on function types

Ordinary function types depends only on it signature.

```
using Fa = int(int,int);
using Fb = int(int,int);
static_assert(is_same_v<Fa,Fb>);
```

When adding contracts, the type declaration becomes unique, much like the type of a lambda.

```
using Fc = int(int a,int) pre(a < 4);
using Fd = int(int a,int) pre(a < 4);
static_assert(not is_same_v<Fc,Fd>);
```

Functions with contracts that want to be of the of the same type need to refer to the same type declaration.

2.3 Defining function from a type

Requiring that functions and pointer who want to have the same type refer to the same declaration has a problem. There are no syntax in C or C++ that allows defining a function from a predefined type.

This proposal will attempt to change that.

The syntax suggested will be that of captureless lambdas. Similar syntax has already been proposed by [P2826R2]. It can be interpreted as lambdas implicitly converts to a compatible function type, but for now it is only allowed for lambdas defined in-place.

Defining

```
using Fa = int(int a,int) pre(a < 4);
Fa fa = [](auto a,auto b) { return a+b; }
```

`a` and `b` are already typed to `int` by Fa so any typing in the argument list must match or be auto. Since the types are already know it may also be possible to allow skipping the type altogether and only give a name.

```
Fa fa = [](auto a,auto b) { return a+b; } // Ok
Fa fa = [](int a,int b) { return a+b; } // Ok
Fa fa = [](auto... a) { return a+...; } // Ok
Fa fa = [](a,b) { return a+b; } // Ok?
Fa fa = [](float a,float b) { return a+b; } // Error
```

2.4 Function pointers

Function pointers can now be created with a type containing contracts. Also, in functions, only the contracts belong to the type will contribute when taking the address of a function.

```
using Fa = int(int a,int) pre(a < 4);
Fa fa = [](a,b) { return a+b; }
// Function pointer
Fa* pa = &fa;
```

Trying to assign the address to a pointer without contracts in the type will fail.

```
using Fb = int(int a,int);
Fb* pb = &fa;  // Error, not the same function type
```

Assigning pointers of different types is not allowed but can be done using captureless lambdas.

2.5 Implementation contracts

Contracts on the type are safe and nice, but it requires rewriting code using this type whenever contracts change. And there are other reasons to have contracts only on the specific function. These contracts is written on the implementation part.

```
using Fa = int(int a,int) pre(a < 4);
Fa fa = [](auto a,auto b) post(b<13) { return a+b; }
```

`fa` now has two contracts, a precondition from the type and a postcondition from the function instance. For a caller that calls fa directly both of these are visible.

Taking the address of a function creates a function pointer with the same type, pointing to a thunk where the implementation contracts are checked.

Contract will be checked in order Type preconditions, implementation preconditions, function body, implementation postconditions, type postconditions.

The possibility to choose if contracts are typed or not let users decide how strict they want their contracts to be applied.

2.6 Old style functions

If a contract on a regular function is on the type or its instance can be either way as we now have a way to specify where we want it. Letting it be on the type by default would be safer, but keeping it on the instance part is more like it works today and will not break code. So this is probably the way to go.

```
int fc(int a,int b) pre(b<13) { return a+b; }
```

Will be equal to:

```
using __FC = auto (int,int) -> int;
__FC fc = [](a,b) pre(b<13) { return a+b; }
```

and can be assigned to a pointer without contracts

```
int (pc*)(int,int) = fc;
```

2.7 Declaring a function type

So far the examples has declared anonymous types referenced by a using or a function declaration. These cannot be used to forward declare a name since they are not unique or may alias.

To have named declaration we need a new keyword identifying function types. Naming such keyword is hard since the names related to function are quite common in code. For now my best suggestion is the composite name `function class`

```
function class F = int(float,int) pre(b==4);
```

When create this way, it will always create unique function types, even without contracts

```
function class Fc = int(int a,int);
function class Fd = int(int a,int);
static_assert(not is_same_v<Fc,Fd>);
```

These can now be safely forward declared as

```
function class F;
function class Fc;
function class Fd;
```

Common practice today is to name function types with a using or a using statement. This can still be allowed, and will create an anonymous type if needed in the same way lambdas do. The only thing the named function types do that this cannot, is the forward declaration of types. For function types without contracts this is still the preferred was as it will still be a non-unique structural type.

```
using Fa = int(int a,int);
using Fb = int(int a,int);
static_assert(is_same_v<Fa,Fb>);
```

2.8 Templated function types

A template can result in a type with contracts either by using a templated function type or by being used to initiate a typed function

```
template<typename E>
using Fe = int (int,E e) pre(e < 4.14);
```

```
template<typename E>
Fe<E> fb = [](a,b) { return a+b; };

fb(A{},3.14); // Has type Fe<double>

Fe<int> fi = fb; // Instantiate for int under name fi

// Used in a generic lambda
auto fg = []<typename T>Fe<T>(a,b) { return a+b; };
```

2.9 Function declarations

Forward declarations will have a new form that do not match forward declaration of variables not old style functions. This is because functions may have instance properties that refer to the arguments, so these must be named.

```
using Fa = int(float,int);

Fa fa;
int fa(float,int);
int fa(float,int b) pre(b==4);
Fa fa = [](float a,int b) pre(b==4);
Fa fa = [](auto a,auto b) pre(b==4);
```

When having a contracted type the named typed must be used in the forward declaration

```
using Fc = int (int a,int) pre(a < 4);

Fc fc = [](a,b) pre(b ==4);
Fc fc;

using Fd = decltype(fd)

Fd fd = [](a,b) pre(b ==3);
```

2.10 Instantiating a lambda from a type

```
auto fb = []Fb(a,b) { return a+b; }; // New syntax
```

This is needed to get the correct type when decaying this to a function pointer

```
Fb* fp = fb; // Ok
```

2.11 Member functions

Member function types are messy.

```
struct X {
    using Xf = int (int);
    using Zf = int (this X, int, int);
    using Xg = int (int) pre (x==3);  // Error, no member access
    using Xg = int (this X,int) pre (x==3);  // Ok

    int x;
    static Xf my_static_function;
    Xf my_member_function;
};

X::Zf X::* pc;
pc = &X::my_member_function;
X::Xf * ps;
ps = &X::my_static_function;
```

The `Xf` type can be both a free function and a member depending on context. This is problematic when adding contracts since it is not clear where to do the lookup for non argument contracts. Therefore it is proposed that only function types with a explicit this argument may refer to members in its contracts Non-static member functions should use the explicit this or trailing const versions.

The function type syntax need to be extended to allow a this parameters to be used in function types to denote a member function type, as this strangely seems to not have been introduced together with deducing this.

Since a free function and a member function can be created from the same function type, a wrapper class can be defined

```
template<typename F>
struct Func {
    F* f;
    F operator() = [](auto... a) { return f(std::forward<decltype(a)>(a)...); }
};
```

where the call operator will have the same contracts as the type F.

2.12 Virtual functions

Virtual functions with contracts on their types works the same way as virtual functions specified in [P2900R12]. Ideally the type contracts should be considered the interface contracts, and the check procedure be:

1. Base type precondition
2. Derived implementation precondition
3. Derived implementation postcondition
4. Base type postcondition

But a version compatible with the existing [P2900R12] would be

1. Base type precondition
2. Base implementation precondition
3. Derived implementation precondition
4. Derived implementation postcondition
5. Base implementation postcondition
6. Base type postcondition

3 Conclusion

As has been seen, contracts on function types be used for adding contracts to function pointers. It requires a generalization of some of the established C++ type system to function types, but without to much novel innovation. The fears previously aired about the effects of contracts in types are not there.

4 References

[P2826R2] Gašper Ažman. 2024-03-18. Replacement functions. https://wg21.link/p2826r2

[P2900R12] Joshua Berne, Timur Doumler, Andrzej Krzemieński. 2024-12-17. Contracts for C++. https://wg21.link/p2900r12

[P3327R0] Timur Doumler. 2024-10-16. Contract assertions on function pointers. https://wg21.link/p3327r0