Proposal for C2y

WG14 N3449

Title: Enhanced type variance

Author, affiliation: Christopher Bazley, Arm. (WG14 member in individual capacity – GPU expert.)

Date: 2025-01-06

Proposal category: New features

Target audience: General Developers, Compiler/Tooling Developers

Abstract: Casting function pointers risks undefined behaviour if a called function is incompatible with the type of the expression used to call it. Wrapper functions avoid such casts but undermine the accessibility and expressiveness of the language. This paper proposes a relaxation of the rules for simple assignment to enhance the language's existing support for type variance.

Prior art: N2607.

Enhanced type variance

Reply-to: Christopher Bazley (chris.bazley@arm.com)
Document No: N3449
Date: 2025-01-06

Summary of Changes

N3449

- Initial proposal

Philosophical underpinning

Although new language features are useful, many users of C would be better served simply by permitting translation of programs that they might reasonably expect to be conforming, but which are currently non-conforming for obscure reasons. An example of such remediation is N2607 [1].

One aspect is the language's treatment of pointers to `void`, which is not addressed by this paper; another, addressed here (as well as by N2607), is its treatment of type qualifiers.

In both cases, it has proven impractical to teach programmers the techniques necessary to write code that is both strictly conforming (i.e. portable) and type safe (i.e. verifiable). The scale of this task is monumental. Even when programmers become aware of the issues, it can be difficult to persuade them to expend the required extra effort.

Most programmers instead make the simplest possible change to suppress diagnostic messages: they add casts. Arguably, compiler options such as `-Wsign-conversion` encourage this mindset by making casting commonplace and therefore acceptable. Casts undermine readability and type safety, both of which are more significant to correctness and maintainability than the mostly theoretical matter of strict conformance.

One of the guiding principles of the C standard charter [2] is "Keep the language small and simple":

> *Simplicity enables both programmers and tools to reason about code, allows for diverse implementations, keeps compilation times short, and helps to achieve other principles.*

When this principle serves both implementers and programmers, it is indisputable; when it mainly serves implementers at the expense of programmers, it should be questioned. No one should have to reason about code that is theoretically sound and would perform as expected if translated.

The existing constraints on assignment appear to conflict with another principle of the charter, "Enable secure programming":

> *Software interfaces should be analysable and verifiable. The language should allow programmers to write concise, understandable, and readable code.*

Programs that use casts to override type checking are not verifiable; programs that define additional functions or objects to avoid the need for such casts are not concise, understandable or readable.

In practice, the rigidity of C's type system undermines type safety due to human factors.

Theoretical basis

Variance is the study of how subtyping of complex types (such as function pointers) relates to subtyping of their constituent parts (such as parameters and return values).

There is a common perception that C does not support subtyping. It is true that C does not allow programmers to create their own subtypes in the same way as C++, such that each subclass inherits members of its superclass and upcasting from subclass to superclass type is implicit.

Techniques exist for implementing polymorphism for user-defined types in C, but an explanation of them lies outside the scope of this paper. Instead, this paper will argue that C does effectively already have subtype polymorphism in a limited form through its mechanism of type qualifiers.

A type, Q, is a subtype of another type, R, if an object of type Q can safely be used wherever an object of type R can be used. This is known as substitutability. Subtypes can be substituted for supertypes.

A type, Q, is a subtype of another type, R, if an object of type R can represent all the values of an object of type Q. This is known as subsumption. Supertypes subsume subtypes.

Now let's consider two declarations of typedef names in a C program:

```
typedef int *Q; // pointer to a mutable object
typedef const int *R; // pointer to a mutable or immutable object
```

In a C program, any object can be accessed through a `const`-qualified lvalue, but only mutable objects can be modified through a modifiable lvalue without invoking undefined behaviour[1]. An example of an immutable object is an object defined with a `const`-qualified type. A pointer to an unqualified referenced type which may hold the address of an immutable object is a maintenance hazard.

Type `Q` can only safely represent pointers to mutable objects of type `int`[2]. Type `R` can safely represent pointers to immutable objects of type `int` in addition to the set of pointers to mutable objects of type `int` representable by type `Q`. (`R` subsumes `Q`.)

In assignments or function calls, an expression of the same type as `Q` can safely be assigned to an lvalue or function parameter of the same type as `R`. This is safe because the `const` qualifier on the referenced type of `R` is a restriction on access to the pointed-to object, and assignment to a more-qualified type is safe. If all `const` qualifiers were removed from a program, its behaviour would be unchanged. (`Q` is substitutable for `R`.)

Effectively, a conversion from type `Q` to type `R` is an upcast, which is why it does not cause the translator to produce a diagnostic message. Unfortunately, many other conversions that could be termed upcasts are constraint violations in C.

[1] The ISO C standard also describes 'volatile accesses' to objects which were not necessarily defined with a volatile-qualified type. These are permitted for the same reason as const accesses.

[2] In the context of type theory, the implementation-specific encoding of a pointer isn't relevant. Nor have I considered cases where integer values are converted to pointer types. Type `Q` can also represent a pointer to an immutable object but not safely.

Problem statement

Callback functions are a common pattern in C programs. They can be used to implement polymorphism, for event processing, to signal completion of asynchronous actions, or to execute the same action for each item of a collection.

Within the standard library, examples include the comparison function pointer passed to the `bsearch` and `qsort` functions, and the cleanup function pointer registered by calling the `atexit` function.

The following example [3] illustrates a problem with the compatibility of function pointer types:

```
void printnum(const int *c)
{
    printf("%d", *c);
}

void foreach(size_t n, int a[n], void (*f)(int *c))
{
    for (size_t i = 0; i < n; ++i)
    {
        f(&a[i]);
    }
}

int main(void)
{
    int array[] = {23, 13, 10};
    foreach(3, array, printnum); // constraint violation
    return 0;
}
```

During translation of the above program, implementations are required to produce a diagnostic similar to the following error reported by GCC:

```
<source>: In function 'main':
<source>:19:23: error: passing argument 3 of 'foreach' from
incompatible pointer type [-Wincompatible-pointer-types]
   19 |     foreach(3, array, printnum); // constraint violation
      |                       ^~~~~~~~
      |                       |
      |                       void (*)(const int *)
<source>:8:41: note: expected 'void (*)(int *)' but argument is of
type 'void (*)(const int *)'
    8 | void foreach(size_t n, int a[n], void (*f)(int *c))
      |                                  ~~~~~~~^~~~~~~~~~
```

This is strange, because the type of the `printnum` function is more permissive (for callers) than the type of the expression used to call it. Ultimately, it undermines the usefulness of type qualifiers, since they cannot be used in parameter declarations without knowledge of how such a function may be called in future.

A possible solution [4] is to change the parameter type of the `printnum` function from `const int *` to `int *`, to avoid triggering the error:

```
void printnum(int *c)
{
    printf("%d", *c);
}

void foreach(size_t n, int a[n], void (*f)(int *c))
{
    for (size_t i = 0; i < n; ++i)
    {
        f(&a[i]);
    }
}

int main(void)
{
    int array[] = {23, 13, 10};
    foreach(3, array, printnum); // okay
    return 0;
}
```

This solution makes the type of the `printnum` function more restrictive (for callers); it relies on the assumption that `printnum` has no other callers which might need to pass the address of a `const`-qualified `int`. It also prevents the implementation from producing diagnostics to warn about accidental modification of an `int` whose address is passed to `printnum`.

An alternative solution [5] would be to introduce a wrapper function that exactly matches the type required by `foreach`:

```
void printnum(const int *c)
{
    printf("%d", *c);
}

void printnum_wrapper(int *c)
{
    printnum(c);
}

void foreach(size_t n, int a[n], void (*f)(int *c))
{
    for (size_t i = 0; i < n; ++i)
    {
        f(&a[i]);
    }
}

int main(void)
{
    int array[] = {23, 13, 10};
    foreach(3, array, printnum_wrapper); // okay
    return 0;
}
```

Introducing the printnum_wrapper function allows callers of printnum to continue to pass the address of a const-qualified int, but it is more effort than many programmers would be willing to spend to work around what they may perceive as capricious behaviour by the translator.

A more common solution [6] is to cast the type of the printnum function pointer to force it to match the unqualified type accepted by the foreach function:

```
void printnum(const int *c)
{
    printf("%d", *c);
}

void foreach(size_t n, int a[n], void (*f)(int *c))
{
    for (size_t i = 0; i < n; ++i)
    {
        f(&a[i]);
    }
}

int main(void)
{
    int array[] = {23, 13, 10};
    foreach(3, array, (void (*)(int *))printnum); // not type-safe
    return 0;
}
```

This is risky because it suppresses any diagnostics that might otherwise be produced if the type of printnum and the type of the expression used to call it within the foreach function mismatch more severely. For example, printnum might expect to receive a pointer to double instead of a pointer to int.

Now, let's consider an alternative scenario in which the printnum function was **originally** declared as accepting the address of an unqualified int:

```
void printnum(int *c)
{
    printf("%d", *c);
}
```

To allow callers of printnum to pass the address of a const-qualified int and to guard against accidental modification of the referenced object, a programmer may want to add a const qualifier to the parameter declaration. This is impossible without breaking any existing code which assigns the address of the printnum function to a pointer, even though the addition of a const qualifier makes use of printnum more permissive (for callers).

This issue hinders modification of existing functions to add the new qualifier proposed by N3422 [7] and is also a barrier to wider adoption of the qualifiers introduced by ANSI C many decades ago. Since qualifiers are a core part of type safety in C, the current situation is untenable.

Analysis

Shortcomings in the current rules for compatibility of function pointers, as they relate to qualifiers, can be generalised into three cases:

Contravariance of parameter types

Example [8]:

```
// The parameter type of foo is more permissive than needed by *bar
int foo(const int *);
int (*bar)(int *) = &foo;
```

This is a constraint violation because C treats function pointer types as invariant.

In contrast, a direct call to the function `foo` is not a constraint violation [9]:

```
// The parameter type of foo is more permissive than needed by bar
int foo(const int *);
int bar(int *x)
{
   return foo(x);
}
```

C allows an expression of type `int *` to be assigned to an lvalue of type `const int *`. As previously expounded, type `int *` is effectively a subtype of `const int *` because a pointer to a mutable object can be substituted for a pointer to a mutable or immutable object of the same type.

(A cat is an animal; therefore, a cat can be used wherever an animal is required. Any specialized abilities of cats are irrelevant.)

The relationship between the simple types is reversed for function types with differently qualified parameters: `int (*)(const int *)` should be a subtype of `int (*)(int *)` because a function that accepts a pointer to a mutable or immutable object can be substituted for a function that only accepts a pointer to a mutable object. This reversal is called contravariance.

(If a function accepts any animal, it can accept a cat.)

It should follow that a function pointer of type `int (*)(const int *)` can be assigned to an lvalue of type `int (*)(int *)`, but it cannot.

Covariance of return types

Example [10]:

```
// The return type of baz is more permissive than needed by *qux
int *baz(int *);
const int *(*qux)(int *) = &baz;
```

This is a constraint violation because C treats function pointer types as invariant.

In contrast, a direct call to the function `baz` is not a constraint violation [11]:

```
// The return type of baz is more permissive than needed by qux
int *baz(int *);
const int *qux(int *x)
{
   return baz(x);
}
```

For function return values, the relationship between the simple types is **not** reversed: `int *(*)(int *)` should be a subtype of `const int *(*)(int *)` because a function that returns a pointer to a mutable object can be substituted for a function that returns a pointer to a mutable or immutable object. This is called covariance.

(If a function is required to produce an animal, it can legitimately produce a cat.)

It should follow that a function pointer of type `int *(*)(int *)` can be assigned to an lvalue of type `const int *(*)(int *)`, but it cannot.

Variance of both parameter types and return types

The relationship between two function types can be covariant for their return types but contravariant for their parameter types.

Example [12]:

```
// The return and parameter types of snap are more permissive
// than needed by *pop
int *snap(const int *);
const int *(*pop)(int *) = &snap;
```

This is a constraint violation because C treats function pointer types as invariant.

In contrast, a direct call to the function `snap` is not a constraint violation [13]:

```
// The return type and parameter types of snap are more permissive
// than needed by pop
int *snap(const int *);
const int *pop(int *x)
{
   return snap(x);
}
```

`int *(*)(const int *)` should be a subtype of `const int *(*)(int *)` because the two function types are contravariant on their parameter types and covariant on their return types.

It should follow that a function pointer of type `int *(*)(const int *)` can be assigned to an lvalue of type `const int *(*)(int *)`, but it cannot.

Subtype relationships

Types on the left can be substituted for (i.e. assigned to) types on the right; types on the right can represent all values of types on the left. These principles should not be taken to imply that substitutions in the text of a program would be beneficial.

- `int *` is effectively a subtype of `const int *`. It can only represent a pointer to a mutable object, which can be treated as immutable (by making only const accesses).
- `void (*)(const int *)` should be a subtype of `void (*)(int *)`. It accepts a pointer to a mutable or immutable object; therefore, it can accept a pointer to a mutable object and treat it as immutable.
- `int *(*)(void)` should be a subtype of `const int *(*)(void)`. It returns a pointer to a mutable object, which can be treated as immutable.
- `int *(*)(const int *)` should be a subtype of `const int *(*)(int *)`. It accepts a pointer to a mutable or immutable object; therefore, it can accept a pointer to a mutable object and treat it as immutable. It returns a pointer to a mutable object, which can be treated as immutable.

Of these relationships, only the first is currently supported by ISO C.

Why does so little of this work?

When a function is called, section 6.5.2.2 of the ISO C standard mandates that:

> *The number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter*

The phrase "such that its value may be assigned to an object" means that we need to consult the rules for simple assignment in section 6.5.17.2.

The most relevant constraint is:

> *— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand;*

Derived types such as array, structure, function and pointer types can be constructed recursively (e.g., an array of pointers to functions returning `int`) but the standard is not explicit about whether "qualified or unqualified versions" and "all the qualifiers of the type" are meant to apply recursively.

A definition of compatible type is given in section 6.2.7:

> *Two types are compatible types if they are the same. Additional rules for determining whether two types are compatible are described in 6.7.3 for type specifiers, in 6.7.4 for type qualifiers, and in 6.7.7 for declarators*

The semantics of type qualifiers given in section 6.7.4 include:

> *For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.*

Consequently, functions cannot be called via pointer types that are not identically qualified (6.3.3):

> *A pointer to a function of one type can be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.*

Section 6.2.5 implies there is no technical barrier to use of a differently qualified version of the same type in scenarios where a translator could not feasibly output machine instructions to perform a conversion (e.g., because the type is part of a derived type such as a pointer to a pointer):

> *The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.*

Also:

> *Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements.*

In practice, translators allow [14]:

```
const int *l = NULL; // pointer to a mutable or immutable object
int *r = NULL; // pointer to a mutable object
l = r; // implicit upcast
```

And [15]:

```
const int *l = NULL; // pointer to a mutable or immutable object
const int *r = NULL; // pointer to a mutable or immutable object
l = r;
```

But not [16]:

```
int *l = NULL; // pointer to a mutable object
const int *r = NULL; // pointer to a mutable or immutable object
l = r; // constraint violation: implicit downcast
```

Or [17]:

```
const int **l = NULL; // pointer to a pointer to a mutable or
immutable object
int **r = NULL; // pointer to a pointer to a mutable object
l = r; // constraint violation
```

We can therefore infer that `const int *` and `int *` are not considered to be "qualified or unqualified versions of compatible types" for the purpose of an assignment in which the operands have pointer type, despite having the same representation and alignment requirements.

Double indirection

It may not be obvious why an expression of type `int **` cannot be assigned to an lvalue of type `const int **`, given that `int *` can be assigned to `const int *`.

In both cases, the `const`-qualified referenced type can safely represent a superset of object addresses representable by the unqualified referenced type. The problem is that a double-indirected pointer can be used to modify two objects: the pointer it points to, and the object pointed to by that pointer.

An example is provided in the ISO C standard to justify the constraints on simple assignment [18]:

```
void foo(void)
{
    const char **cpp;
    char *p;
    const char c = 'A';
    cpp = &p; // constraint violation
    *cpp = &c; // valid
    *p = 0; // undefined behaviour (c = 0)
}
```

The type of `cpp` says it is a pointer to a pointer to a mutable or immutable `char`. After `*cpp` has been made into an alias for `p` (thereby violating a constraint), `*cpp` can be modified to make `*p` alias `c`, an immutable `char`. This circumvents checking of the type of `p`, which can only safely point to a mutable `char`.

The same example can be adapted to use a function call, with similar effect [19]:

```
const char c = 'A';

void bar(const char **cpp)
{
    *cpp = &c; // valid
}

void foo(void)
{
    char *p;
    bar(&p); // constraint violation
    *p = 0; // undefined behaviour (c = 0)
}
```

It follows that `void (*)(const char **)` is not substitutable for `void (*)(char **)` because a function of the first type can easily assign the address of an immutable object to the pointer whose address was passed by the caller; such assignments are unsafe if the referenced pointer has a non-`const`-qualified referenced type.

The whole problem can be avoided by qualifying the pointed-to pointer, i.e. using type `const char *const *` instead of `const char **`.

That modification increases the number of constraint violations in comparison to the original example [20]:

```
void foo(void)
{
    const char *const *cpcp; // 2nd const makes first CV redundant
    char *p;
    const char c = 'A';
    cpcp = &p; // constraint violation
    *cpcp = &c; // constraint violation (p = &c)
    *p = 0; // undefined behaviour (c = 0)
}
```

This shows there is no need to forbid assignment of an expression of type `char **` to an lvalue of type `const char *const *` (which is the first constraint violation). With those types, the assignment `cpp = &p` can be permitted without creating an opportunity to circumvent checking of the type of `p`. `*cpp` still aliases `p` but `*cpp` can no longer be used to modify `p`.

For every extra level of indirection, an extra `const` qualifier is required to make it safe to permit the first assignment [21]:

```
void foo(void)
{
    const char *cp, *const *const *cpcpcp;
    char **pp;
    const char c = 'A';
    cpcpcp = &pp; // constraint violation
    cp = &c;
    *cpcpcp = &cp; // constraint violation (pp = &cp)
    **pp = 0; // undefined behaviour (*cp = 0, c = 0)
}
```

If the chain of indirection includes any array types, their element types must likewise be `const`-qualified to prevent them being used to circumvent type-checking of pointers further along the chain.

It is important not to draw the wrong conclusion about recursively qualifying derived types. Each pointed-to pointer isn't qualified as `const` because its referenced type is `const`-qualified; it is `const`-qualified to prevent it being modified to point to an object that has all the qualifiers of its referenced type.

Examples using `volatile` instead of `const` should help to clarify this:

- `int **` should be subtype of `volatile int *const *`.
- `void (*)(volatile int *const *)` should be a subtype of `void (*)(int **)`.
- `int **(*)(void)` should be a subtype of `volatile int *const *(*)(void)`.
- `int **(*)(volatile int *const *)` should be a subtype of `volatile int *const *(*)(int **)`.

Function types are special because a function's return value is not an lvalue; there is no point qualifying a return type as `const`, since it cannot be assigned to [22]:

```
void foo(void)
{
    const char *const (*cpcfp)(void); // 2nd const is redundant
    char *pf(void);
    const char c = 'A';
    cpcfp = &pf; // constraint violation
    (*cpcfp)() = &c; // lvalue required as left operand
    *pf() = 0; // valid
}
```

Because the return value of `(*cpcfp)()` is merely a copy of the result of evaluating some expression, assigning to it cannot modify any object to point to `c`; nor does calling the same function again as `pf()` yield a pointer to `c`.

Qualifiers on return types are treated inconsistently by GCC 14.2.0 and Clang 18.1.0: both warn about such qualifiers in declarations and claim to ignore them, but Clang halts if a return type is qualified in the right operand of a simple assignment but not the left operand [23].

It may be necessary to `const`-qualify function pointers to prevent them being used to circumvent type-checking of pointers further along the chain [24]:

```
const char c = 'A';

const char *cpf(void)
{
    return &c;
}

void foo(void)
{
    const char *(*const *cpfpcp)(void);
    char *(*pfp)(void);
    cpfpcp = &pfp; // constraint violation
    *cpfpcp = &cpf; // constraint violation (pfp = &cpf)
    *(*pfp)() = 0; // undefined behaviour
}
```

The return type of `**cpfpcp` does not need to be qualified (as `const char *const`) to make it safe to permit the first assignment. However, `*cpfpcp` must be `const`-qualified to prevent it being modified to make `*pfp` alias `cpf`, a function that returns a pointer to an immutable `char`.

Output parameters

Indirection is often used to allow a function to have more than one return value, by requiring callers to provide a pointer to an object to be modified [25]:

```
#include <stdio.h>

int divide(int dividend, int divisor, int *remainder)
{
    *remainder = dividend % divisor;
    return dividend / divisor;
}

void test(int (*divider)(int, int, int *))
{
    int remainder;
    int quotient = (*divider)(6, 4, &remainder);
    printf("quotient=%d, remainder=%d.\n", quotient, remainder);
}

int main(void)
{
    test(&divide);
    return 0;
}
```

This paper proposed that `void (*)(const int *)` should be a subtype of `void (*)(int *)` because it is of no consequence to callers of a function with the latter signature that their callee could also accept the address of an immutable object.

However, if a function of type `int (int, int, const int *)` were substituted for the `divide` function in the example above, the substitute function could not (without casting) assign a value to the object whose address is passed to it by `test`. Consequently, the value of `remainder` would still be indeterminate at the point where it is printed.

It would be mistaken to attribute this bug entirely to variance: the type of the `divide` function does not guarantee that it will assign a value to `remainder` on every path of execution; nor does the alternative signature `int (int, int, const int *)` guarantee that a function of that type will never assign a value to `remainder` (although it makes it unlikely).

The real issue is that the language lacks syntax to distinguish input parameters from output parameters: a translator cannot tell that the third parameter is intended for output, so it cannot tell that a substitution might be inappropriate. That doesn’t invalidate the concept of variance for function types, but it does illustrate a potential drawback.

A more robust caller of the `divide` function would pre-initialise `remainder`:

```
int remainder = 0;
int quotient = divide(6, 4, &remainder);
printf("quotient=%d, remainder=%d.\n", quotient, remainder);
```

This makes it more likely that the program will behave predictably even though its division interface is not analysable and may not have been implemented correctly.

An alternative solution would be to return the quotient and remainder as members of a `struct` type [26]:

```
#include <stdio.h>

typedef struct
{
    int quotient, remainder;
} divide_result;

divide_result divide(int dividend, int divisor)
{
    return (divide_result){
        .quotient = dividend / divisor,
        .remainder = dividend % divisor
    };
}

void test(divide_result (*divider)(int, int))
{
    divide_result res = (*divider)(6, 4);
    printf("quotient=%d, remainder=%d.\n",
           res.quotient, res.remainder);
}
```

This avoids the possibility of accidentally using an indeterminate value. In some circumstances, it may also be more efficient than using an output parameter. For example, ARM64 GCC 14.2.0 packs the 32-bit members of the `struct` type into a single 64-bit register, which avoids memory accesses at the expense of slightly lower code density.

Output parameters can also have pointer types, which entails double indirection [27]:

```
void alloc_cint(const int **out)
{
    static const int p;
    *out = &p;
}

void whelk(void)
{
    const int *r;
    alloc_cint(&r);
}
```

If a parameter is a pointer to a pointer to a qualified type (as in the above example), then the pointer to the qualified type must be `const`-qualified (unlike in the above example) to allow variance for function pointer types as previously described.

The signature of the function used in the previous example could be modified as follows:

```
void alloc_cint(const int *const *out);
```

Contravariance of parameter types says that the address of the function could now be assigned to a pointer of type `void (*)(int **)`, but the function has also become useless because it can no longer assign a value to `*out`.

The only kind of variance that makes sense for output parameters is covariance (as for return types). According to covariance, `void (*)(int **out)` should be a subtype of `void (*)(const int **out)` just as `int *(*)(void)` is a subtype of `const int *(*)(void)`. This is the opposite of the subtyping relationship for ordinary parameters.

It would be unsafe to assume that all function parameters of pointer type that do not have a `const`-qualified referenced type are output parameters. The caller of a function of type `void (const int **)` can pass the address of a pointer to an object defined as `const`; if it were permissible to substitute a function of type `void (int **)` then the substitute function could modify the object through an lvalue which has non-`const`-qualified type.

Consequently, covariance cannot be supported for output parameters.

Luckily, it is rare to want to substitute a function that returns pointer to a **less**-qualified result. However, lack of covariance for output parameters does also mean that a function pointer type cannot restrict access to an object whose address is returned via an output parameter, beyond whatever restrictions the pointed-to function imposes.

Function pointers as parameters

Parameters can be pointers to other functions, including recursively.

In the following declaration, `foo` may be the address of a function that allows its caller to pass the address, `bar`, of a second function that allows its caller to pass the address, `qux`, of a third function:

```
void (*foo)(int *(*bar)(char *(*qux)(void)));
```

One or more return types (e.g., `char *`) may be part of a parameter type (e.g., `char *(*qux)(void)`), without any obvious separation between them. This is potentially confusing because a parameter type describes a source, whereas a return type describes a sink (regardless of whether that sink is also part of the description of a source).

This paper argued that `int *(*)(const int *)` should be a subtype of `const int *(*)(int *)`. According to contravariance of parameter types, the relationship between a subtype and its supertype should be reversed when the type is used as a parameter.

Contravariance applies most straightforwardly to the chain of type derivations leading from a function type to the outermost derivation of the parameter type that incorporates it. For example, `void (*)(void (**p)(void))` should be a subtype of `void (*)(void (*const *p)(void))` because the latter function can treat the object designated by `*p` as immutable regardless of whether it is or not.

Applying contravariance to a function type in a parameter type is trickier. It would be nonsense to think that a function's parameters become sinks, and its return value becomes a source, just because it is called through a pointer passed as a parameter. Nevertheless, the relationship between subtype and supertype is reversed. How?

Covariance and contravariance are situational: the relationship between parameter types is only contravariant relative to the relationship between the same types in simple assignment. In both cases, a value of one type is (potentially) being assigned to an object of another.

Parameter types only *appear* contravariant in an assignment of a function pointer because a parameter declaration in the type of the left operand describes the converted type of an argument provided by a caller, whereas the type of the right operand describes the type of the same value in the callee.

Argument passing can be modelled as assignment; therefore, the parameter type received by a callee can be characterised as the type of an assignee object and the parameter type passed by its caller as the type of an assigned value:

```
void (*a)(int *assigned);
void b(const int *assignee);
a = &b; // assigned type on the left, assignee type on the right
int assigned;
(*a)(&assigned); // assigned type is reinterpreted as assignee type
```

This is a reversal of the usual position of assigned and assignee types:

```
const int *assignee;
int assigned;
assignee = &assigned;
```

If the `actual` parameter of `a` and `expected` parameter of `b` have function pointer types instead of object pointer types, their positions are still reversed relative to simple assignment:

```
void (*a)(void (*assigned)(const int *));
void b(void (*assignee)(int *));
a = &b; // assigned type on the left, assignee type on the right
void a2(const int *); // can accept mutable objects too
(*a)(&a2); // assigned type is reinterpreted as assignee type
```

When `*a` is called with the address of `a2`, the function executed is `b`, which has parameter type `void (*)(int *)`. If `b` calls `a2` then it does so according to the parameter type of `b`, not the parameter type of `*a` (which would be `void (*)(const int *)`). Hence, the type of the pointer that `b` passes to `a2` is `int *` (part of the *assignee* type in the call to `*a` but acting as *assigned* type in the call to `a2`) rather than `const int *` (part of the *assigned* type in the call to `*a` but acting as *assignee* type in the call to `a2`).

The above declarations could be rewritten to emphasise the call to the function whose address is passed as a parameter, instead of the call to the top-level function:

```
void (*a)(void (*)(const int *assignee));
void b(void (*)(int *assigned));
a = &b; // assignee type on the left, assigned type on the right
```

Given that `int *` can also be assigned directly to `const int *`, this could be read as a refutation of contravariance for parameter types; instead, it shows that the designation of parameter types as assignee or assigned type is swapped at every nesting level:

```
void (*assignee)(void (*assigned)(const int *assignee));
void assigned(void (*assignee)(int *assigned));
assignee = &assigned;
```

The following example illustrates contravariance of the type of a parameter, bazpp, that is a pointer to a pointer to a function [28]:

```
#include <stdio.h>

// *h is immutable although foo and bar don't require that
int *baz(const int *h)
{
    // i is mutable although foo and bar don't require that
    static int i;
    i = *h;
    return &i;
}

const int *bar(const int *(*const *bazpp)(int *h), int *h)
//             ^^^^^^^^^^^^ variance ^^^^^^^^^^^^
{
    /* It would not be safe to allow *bazpp to be modified,
       e.g. it could be used to modify bazp to point to
       a function returns a pointer to an immutable int.
    */
    return (**bazpp)(h);
}

void foo(void)
{
    const int *(*barp)(int *(**bazpp)(const int *h), int *h) = &bar;
    //                       ^^^^^^^^ variance ^^^^^^^^^^
    int h = 5;
    int *(*bazp)(const int *) = &baz;
    const int *i = (*barp)(&bazp, &h);
    printf("%d", *i);
}
```

In the above example, let fn_t be a polymorphic type resembling int *(int *). If the assignment of &bar to barp is permitted then fn_t *const * (the type of the bazpp parameter of bar) is substituted for fn_t ** (the type of the same parameter of *barp). This is contravariance.

In this case, const-qualification of *bazpp is not only permitted but *required* because the pointed-to type fn_t is variant in its parameter type (const int * is substituted for int *) and its return type (int * is substituted for const int *).

It would be unsafe to allow bar to modify *bazpp because it aliases bazp. If *bazpp could be modified to point to a function that accepts only mutable objects (passed by reference) and returns a pointer to an immutable object, then calling such a function through *bazp would be unsafe.

(This was previously discussed in the section on double indirection.)

Function pointers as return values

Return values can be pointers to other functions, including recursively.

In the following declaration, foo may be the address of a function that returns the address of a second function that accepts the address of an int and returns the address of a third function that accepts the address of a char and returns the address of a double:

```
double *(*(*(*foo)(void))(int *))(char *);
```

Since C23, the same declaration may alternatively be written as:

```
typeof(typeof(typeof(double *(char *)) *(int *)) *(void)) *foo;
```

This paper argued that int *(*)(const int *) should be a subtype of const int *(*)(int *). According to covariance of return types, it should be possible to extend this kind of variance to all the function pointer types derived from double.

For example:

```
const double *(*(*(*a)(void))(int *))(char *);
double *(*(*b(void))(const int *))(const char *);
a = &b; // requires variance
int q;
char r;
double s = *(*(*(*a)())(&q))(&r);
```

A potential complication arises in the case of double-indirection:

```
const double *(*const *
                (*const *
                  (*a)(void)
                )(int *)
              )(char *);
double *(**(**b(void))(const int *))(const char *);
a = &b; // requires variance
int q;
char r;
double s = *(**(**(*a)())(&q))(&r);
```

It is necessary to const-qualify the pointer to the function of type const double *(char *) in the type of a to prevent it being used to circumvent type-checking:

```
const double *unsafe(char *);
*(**(*a)())(&q) = &unsafe; // replaces a double *(const char *)
```

The above assignment must not be allowed because unsafe requires a mutable char to be passed (by reference) and returns the address of an immutable double, neither of which can be assumed according to the type of b. Because of double indirection, the pointer to the third function must exist outside the temporary lifetime of the return value of the second function; therefore, we must assume that pointer (of a different type) could be used to call unsafe with the wrong constraints.

Because the type of the third function varies between a and b, it is necessary to const-qualify the pointer to the second function in the type of a (whether the second function itself varies or not):

```
const double *(*const *unsafe(int *))(char *);
*(*a)() = &unsafe;
```

Proposed wording

The proposed wording is a diff from the N3299 working draft [29]. Green text is new text, while ~~red~~ text is deleted text.

Since the existing EXAMPLE 3 in section 6.5.17.2 illustrates a constraint, it might be misplaced. I have not moved it, nor the new examples, for fear of making the diff unreadable.

6.2.10 Substitutable type

1 A substitutable type is the referenced type of an expression of pointer type which can be assigned to an lvalue of a different pointer type. One type can be substitutable for multiple types.

Additional rules for determining whether one type is substitutable for another apply to function types in a recursively constructed derived type. Let N equal the nesting level of a function type when function type derivations are counted (starting at 1) from innermost to outermost derivation.

A type, Q, is substitutable for another type, R, if the following conditions are satisfied:

- Q and R would be compatible types if all qualifiers except `_Atomic` were removed from both types (including from parameters of function types and from every derivation of a derived type).
- For every referenced type[3] from which a pointer is derived (including recursively) or the type itself if the type consists of no derived types:
  - the type in R has all the qualifiers of the type in Q; and,
  - the type in Q has an atomic type if the type in R has an atomic type; and,
  - the type in R is const-qualified if it is a pointer type derived (including transitively through other derivations[4]) from a referenced type for which Q does not have all the qualifiers of the type in R; or,
  - the type in R is const-qualified if it is a pointer type derived (including transitively through other derivations) from a function type that has at least one parameter of pointer type derived (including transitively) from a referenced type for which Q does not have all the qualifiers of the type in R.
- For every function type derivation that has an odd value of N:
  - the referenced type of every parameter of pointer type satisfies the preceding requirements for substitutability with Q and R swapped; and,
  - the return type does not have a pointer type, or the referenced type of the return type satisfies the preceding requirements for substitutability.
- For every function type derivation that has an even value of N:
  - the referenced type of every parameter of pointer type satisfies the preceding requirements for substitutability; and,
  - the return type does not have a pointer type, or the referenced type of the return type satisfies the preceding requirements for substitutability with Q and R swapped.

This method of determining whether one type is substitutable for another is applied recursively: whether a type that includes function type derivations is substitutable depends on whether referenced types of parameters of those function types are substitutable. Those parameter types may themselves include function type derivations.

[3] A return type is not a referenced type but the function type derived from it may be a referenced type.

[4] This transitive dependency includes function type derivations. A return type can be a pointer type whose referenced type in Q lacks qualifiers that are present in R, or which points to such a type.

6.3.3.3 Pointers

8 A pointer to a function of one type can be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not ~~compatible with~~ substitutable for the referenced type, the behavior is undefined.

6.5.17.2 Simple assignment

Constraints

1 One of the following shall hold:[111)]

- the left operand has atomic, qualified, or unqualified arithmetic type, and the right operand has arithmetic type;
- the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right operand;
- the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to ~~qualified or unqualified versions of~~ compatible types, ~~and the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand~~, or the referenced type of the right operand is substitutable for the referenced type of the left operand;

Semantics

6 EXAMPLE 3 The following fragment ~~can be used as an example~~ illustrates why assigning an expression of pointer type to an lvalue that has a more-qualified referenced type at some level other than its outermost derivation can be unsafe:

```
const char **cpp;
char *p;
const char c = 'A';
cpp = &p; // constraint violation because *cpp is not const
*cpp = &c; // valid because *cpp is not const
*p = 0; // ~~valid~~undefined behavior
```

The first assignment is unsafe because it would allow the following ~~valid~~ code to attempt to change the value of the const object `c`. Because the referenced type of `cpp` is not const-qualified, `*cpp` can be modified to make `p` point to `c` without requiring the expression `&c` to be cast (as would be required if `&c` were assigned directly to `p`).

7 EXAMPLE 4 The following fragment illustrates how assigning an expression of pointer type to an lvalue that has a more-qualified referenced type at some level other than its outermost derivation can be safe:

```
const char *const *cpp;
char *p;
const char c = 'A';
cpp = &p; // valid because *cpp is const
*cpp = &c; // constraint violation because *cpp is const
*p = 0; // undefined behavior
```

The second assignment is unsafe because it would allow the following code to attempt to change the value of the const object `c`. Because the referenced type of `cpp` is const-qualified, `*cpp` cannot be modified to make `p` point to `c` without requiring the expression `&c` to be cast.

8 EXAMPLE 5 The following fragment illustrates assignments involving different pointer types:

```
volatile int vi, *pvi = &vi, **ppvi = &pvi,
             *const *pcpvi,
             ***pppvi,
             *const **ppcpvi,
             **const *pcppvi,
             *const *const *pcpcpvi;

int i, *pi = &i, **ppi = &pi, ***pppi = &ppi;

pvi = pi; // 1. valid
pi = pvi; // 2. constraint violation because *pi is not volatile

ppvi = ppi; // 3. constraint violation because *ppvi is not const
pcpvi = ppi; // 4. valid
ppi = pcpvi; // 5. constraint violation because **ppi is not
                   volatile */

pppvi = pppi; /* 6. constraint violation because *pppvi and **pppvi
                    are not const */
pcppvi = pppi; /* 7. constraint violation because **pcppvi is not
                     const */
ppcpvi = pppi; /* 8. constraint violation because *ppcpvi is not
                     const */
pcpcpvi = pppi; // 9. valid
pppi = pcpcpvi; /* 10. constraint violation because ***pppi is not
                       volatile */
```

The second, fifth and tenth assignments are unsafe because they would allow access to an object (`vi`) defined with a volatile-qualified type through an lvalue (`*pi`, `**ppi` or `***pppi`) which has non-volatile-qualified type.

The third assignment is unsafe because it would allow following code to modify `pi` to point to a `volatile int` by assigning to `*ppvi`.

The sixth and seventh assignments are unsafe because they would allow following code to modify `pi` to point to a `volatile int` by assigning to `**pppvi` or `**pcppvi`.

The sixth and eighth assignments are unsafe because they would allow following code to modify `ppi` to point to the address of a `volatile int` by assigning to `*pppvi` or `*ppcpvi`.

7 EXAMPLE 6 The following fragment illustrates assignments involving function pointer types:

```
void fpi(int *), (*pfpi)(int *) = &fpi,
     (*pfpvi)(volatile int *),
     fppi(int **), (*pfppi)(int **) = &fppi,
     fppvi(volatile int **), (*pfppvi)(volatile int **) = &fppvi,
     (*pfpcpvi)(volatile int *const *);

int *fpvirpi(volatile int *),
    *(*pfpvirpi)(volatile int *) = &fpvirpi,
    *(*pfrpi)(void),
    **frppi(void), **(*pfrppi)(void) = &frppi,
    **(*pfpcpvirppi)(volatile int *const *);

volatile int *fpirpvi(int *), *(*pfpirpvi)(int *) = &fpirpvi,
             *(*pfrpvi)(void),
             **(*pfrppvi)(void),
             *const *frpcpvi(void),
             *const *(*pfrpcpvi)(void) = &frpcpvi,
             *const *fppirpcpvi(int **),
             *const *(*pfppirpcpvi)(int **) = &fppirpcpvi;

pfpvi = pfpi; /* 1. constraint violation because referenced type of
                    parameter of *pfpvi is volatile */
pfpi = pfpvi; // 2. valid

pfrpi = pfrpvi; /* 3. constraint violation because referenced type
                      of return type of *pfrpi is not volatile */
pfrpvi = pfrpi; // 4. valid

pfpvirpi = pfpirpvi; /* 5. constraint violation for the same reasons
                           as 1 and 3 */
pfpirpvi = pfpvirpi; // 6. valid

pfpcpvi = pfppi; /* 7. constraint violation because referenced type
                       of referenced type of parameter of *pfpcpvi
                       is volatile */
pfppi = pfppvi; /* 8. constraint violation because referenced type
                      of parameter of *pfppvi is not const */
pfppi = pfpcpvi; // 9. valid

pfrppvi = pfrppi; /* 10. constraint violation because referenced
                         type of return type of *pfrppvi is not
                         const */
pfrppi = pfrpcpvi; /* 11. constraint violation because referenced
                          type of referenced type of return type
                          of *pfrppi is not volatile. */
pfrpcpvi = pfrppi; // 12. valid

pfpcpvirppi = pfppirpcpvi; /* 13. constraint violation for the same
                                  reasons as 7 and 10 */
pfppirpcpvi = pfpcpvirppi; // 14. valid
```

The first, fifth, seventh and 13th assignments are unsafe because they would allow the `fpi`, `fpirpvi`, `fppi` and `fppirpcpvi` functions to access an object defined with a volatile-qualified type through an lvalue which has non-volatile-qualified type.

The third, fifth, 11th and 13th assignments are unsafe because they would allow the `frpvi`, `fpirpvi`, `frpcpvi` and `fppirpcpvi` functions to return the address of an object defined with a volatile-qualified type that its caller could access through an lvalue which has non-volatile-qualified type.

The eighth assignment is unsafe because it would allow the `fppvi` function to assign the address of an object defined with a volatile-qualified type to a pointer of type `int *` whose address was passed by the caller.

The 10th assignment is unsafe because it would allow a caller of the `frppi` function to assign the address of an object defined with a volatile-qualified type to a pointer of type `int *` whose address was returned by that function.

Conclusion

Many of the ideas and code examples discussed in this paper might seem abstruse.

It is not the author's intent that C programmers should be required to learn new rules or master a new knowledge domain. It should not be necessary to study subtyping or any other aspect of programming language theory to write a C program. Where possible, the language's semantics should follow intuitively from its syntax.

No new syntax rules are proposed by this paper, nor any change to the semantics of type qualifiers in strictly conforming programs. The only change proposed is a quiet improvement to the language's existing support for type polymorphism.

Without any effort on their part, this allows programmers to write code that is more verifiable, concise, understandable, and readable. It also makes it possible to add type qualifiers to library interfaces without entailing an unknown number of constraint violations.

The only new idea that some programmers might benefit from learning is that it can be beneficial to const-qualify pointers, since that allows greater type polymorphism. Arguably, that is already good advice because it avoids mistakes such as [30]:

```
void foo(int *i)
{
    *i++; // no effect; maybe (*i)++ was intended
}
```

Acknowledgements

I would like to recognize Martin Uecker for his encouragement.

References

[1] N2607 Compatibility of Pointers to Arrays with Qualifiers (Uecker, 2020)
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2607.pdf

[2] N3280 The C Standard charter
https://www.open-std.org/JTC1/sc22/wg14/www/docs/n3280.htm

[3] Compiler Explorer
https://godbolt.org/z/dfn8r67ec

[4] Compiler Explorer
https://godbolt.org/z/GMK6rsqs8

[5] Compiler Explorer
https://godbolt.org/z/YYTnWcfKb

[6] Compiler Explorer
https://godbolt.org/z/84h1Wochr

[7] N3422 _Optional: a type qualifier to indicate pointer nullability (v2)
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3422.pdf

[8] Compiler Explorer
https://godbolt.org/z/PP5qzYbYP

[9] Compiler Explorer
https://godbolt.org/z/svP9MnzaK

[10] Compiler Explorer
https://godbolt.org/z/PP5qzYbYP

[11] Compiler Explorer
https://godbolt.org/z/r6Gs5cefc

[12] Compiler Explorer
https://godbolt.org/z/1WhY6rG7n

[13] Compiler Explorer
https://godbolt.org/z/4fMEKGj96

[14] Compiler Explorer
https://godbolt.org/z/e9xMx8qTE

[15] Compiler Explorer
https://godbolt.org/z/obWqsG18E

[16] Compiler Explorer
https://godbolt.org/z/f6eGEEPce

[17] Compiler Explorer
https://godbolt.org/z/47EGYhrP4

[18] Compiler Explorer
https://godbolt.org/z/zahYqEdGE

[19] Compiler Explorer
https://godbolt.org/z/7EcqGPY9j

[20] Compiler Explorer
https://godbolt.org/z/jbv63Ezjv

[21] Compiler Explorer
https://godbolt.org/z/4x5s3bPGd

[22] Compiler Explorer
https://godbolt.org/z/KnjW8bYcn

[23] Compiler Explorer
https://godbolt.org/z/qrYKzdbEj

[24] Compiler Explorer
https://godbolt.org/z/aff9qeor9

[25] Compiler Explorer
https://godbolt.org/z/hW7WrMaa8

[26] Compiler Explorer
https://godbolt.org/z/P6exP7oEP

[27] Compiler Explorer
https://godbolt.org/z/ddKM7sW9s

[28] Compiler Explorer
https://godbolt.org/z/49e1335P7

[29] N3299 Working Draft C2y Post C23-Publication
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3299.pdf

[30] Compiler Explorer
https://godbolt.org/z/EanojanzW