N3937: Constexpr function pointer initialisation

Document #:	N3937
Date:	2026-07-15
Project:	Programming Language C

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

Something surprising to me came up in last year's WG14 meeting in Brno: that the following code is not currently legal C as you cannot initialise a constexpr function pointer to any value other than null:

```
int foo(int x);

constexpr struct fnptrs
{
    int (*fnptr)(int);
} fnptrs = { foo };
```

This is legal in C++ since C++ 11, and of course the following code replacing the `constexpr` with `static const` is legal in C:

```
int foo(int x);

static const struct fnptrs
{
    int (*fnptr)(int);
} fnptrs = { foo };
```

The latter, under optimisation, produces identical output to the former for GCC and clang targeting ELF:

```
        mov     rax, qword ptr [rip + foo@GOTPCREL]
```

This is because the address of a function IS known to the compiler at compile time: for ELF, it is the offset of function `foo()` within the Global Offset Table (GOT) for ELF. Let's turn off position independence so we're a 1970s machine:

```
        mov     rax, OFFSET FLAT:"foo"
```

Here the compiler emits a placeholder for 'whatever value `foo` ends up being given'.

What about structures containing function pointers? If I return the address of `struct fnptrs`:

```
        mov     eax, OFFSET FLAT:"fnptrs"
        ret
"fnptrs":
        .quad   "foo"
```

Now it emits into the program binary a constant read-only structure containing a pointer to the `foo()` function. Returning to C++, both the `static const` and `constexpr` editions produce identical output from the compiler: the address of the `foo()` function is considered a compile-time constant.

To explain, whilst it is the assembler or linker which generates the final value in the final emitted program binary, for *the compiler* function pointer values ARE known at compile time: the function `foo()` has a value of placeholder `foo`. The assembler or linker or runtime later turns that into some hard number, but from the compiler's perspective that transformation is immaterial because the value of function pointers IS a compile-time constant from the compiler's perspective.

It therefore makes no sense to not permit in C the initialisation of constexpr function pointer values from non-calculated function pointer values, same as C++ permits for `constexpr`.

A: Proposed wording (minimal case)

This is against [N3886]. Red strike through is removal, underlined green is addition.

In Section 6.7.2 'Storage-class specifiers' paragraph 6:

If an object or subobject declared with storage-class specifier `constexpr` has pointer, integer, or arithmetic type, any explicit initializer value for it shall be null or a pointer to function, an integer constant expression, or an arithmetic constant expression, respectively. If the object declared has real floating type, the initializer shall have integer or real floating type. If the initializer has decimal floating type, the object declared shall have decimal floating type and the conversion shall preserve the quantum of the initializer. If the initializer has real type and a signaling NaN value, the unqualified versions of the type of the initializer and the corresponding real type of the object declared shall be compatible

B: Proposed wording (expansive case)

The assembler/linker substitution of values for function pointers also is applied to global variables, such that this code is handled identically to function pointers:

```
struct Foo;

extern struct Foo foo;

static const struct fooptr
{
    struct Foo *foo;
} fooptrs = { &foo };
```

It would seem amiss to exclude this use case as well, plus this also works in `constexpr` since C++ 11, so:

In Section 6.7.2 'Storage-class specifiers' paragraph 6:

If an object or subobject declared with storage-class specifier `constexpr` has pointer, integer, or arithmetic type, any explicit initializer value for it shall be null or a pointer to function or an object with static or `constexpr` storage class, an integer constant expression, or an arithmetic constant

expression, respectively. If the object declared has real floating type, the initializer shall have integer or real floating type. If the initializer has decimal floating type, the object declared shall have decimal floating type and the conversion shall preserve the quantum of the initializer. If the initializer has real type and a signaling NaN value, the unqualified versions of the type of the initializer and the corresponding real type of the object declared shall be compatible

1 References

[N3886] Meneide, Working Draft
Working Draft
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3886.pdf