N3407
2025-01-13

Pitch for `#dialect` directive

Jakub Łukasiewicz

- Communicates what dialect code was written for
- Standardizes staple part of build systems
- Mitigates negative effect of breaking changes

* for now, primarily addresses the core language; standard library later

Principles to not break old code

- 9. Minimize incompatibilities with C90 (ISO/IEC 9899:1990). It should be possible for existing C implementations to gradually migrate to future conformance, rather than requiring a replacement of the environment. It should also be possible for the vast majority of existing conforming C programs to run unchanged.

  ~ N444

- 14. Migration of an existing code base is an issue. The ability to mix and match C89, C99, and C1X based code is a feature that should be considered for each proposal.

  ~ N1250

- **Ease migration to newer language editions**
  Developers should be able to mix and match code from different language editions. The bulk of existing codebases should be largely accepted by a translator conforming to a newer language revisions, and the programmer's burden to change code just to have it accepted by a conforming translator must be limited.

  ~ N3280

PROBLEM

One’s bug is another’s feature

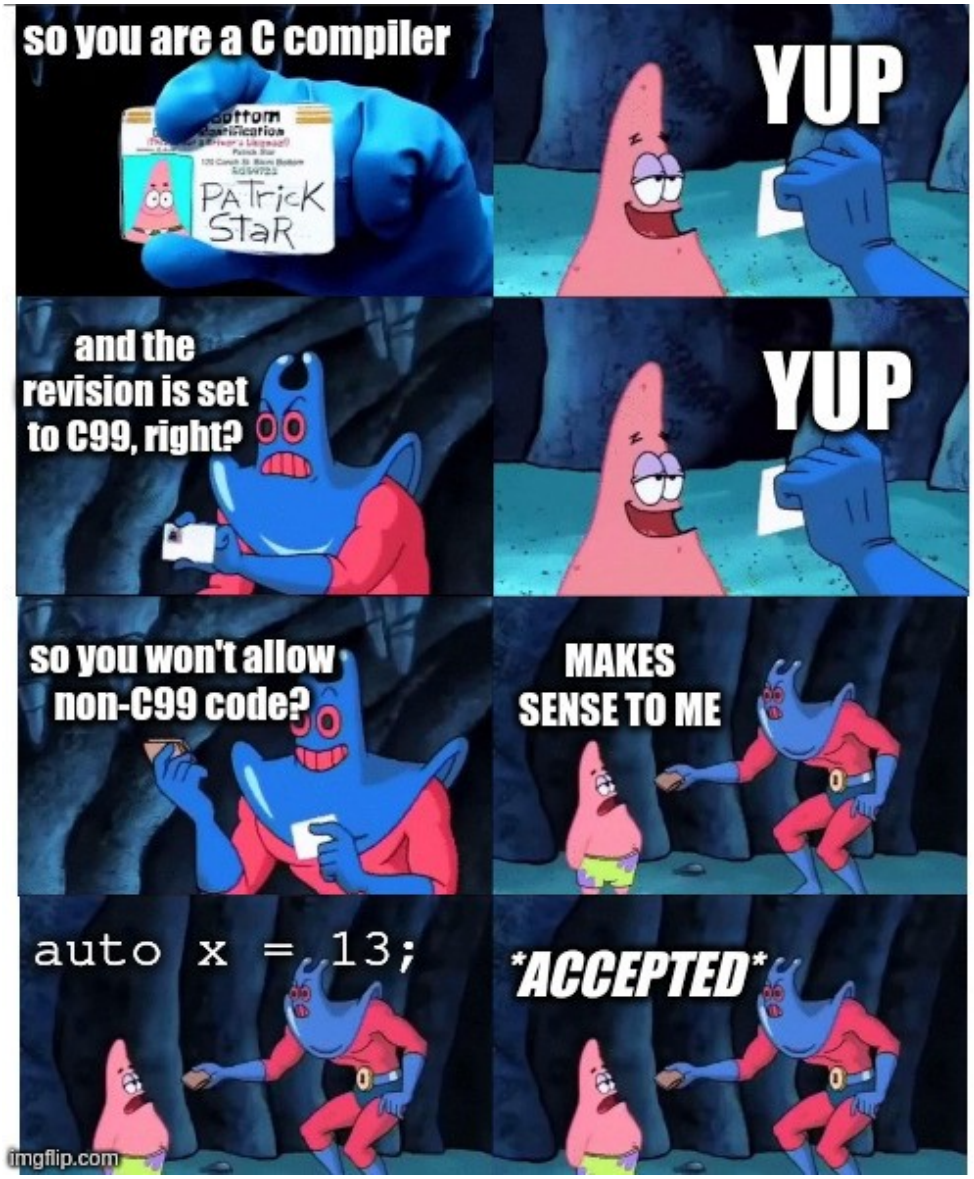
https://xkcd.com/1172/

One's trash is another's treasure

```
struct Ctx { void (*last)(); };
extern void *HIST;
extern void list_append(void*, void (*)());


void bar(int);
void baz(int,int*);
void (*EXCLUDED[100])() = { &bar, &baz };


void hist_append(struct Ctx *ctx)
{
    for (int i = 0; i < 100; ++i) {
        if (ctx->last == EXCLUDED[i]) {
            return;
        }
    }
    list_append(HIST, ctx->last);
}
```

(broken in C23)

On the other hand...



```
$ cat a.c
int main()
{
    auto x = 13;
    return 0;
}
$ gcc -std=c99 a.c
```

Debian < 12	*	success?!
Arch Linux	before May 2024	success?!

It doesn't even say it's been removed!

```
a.c:3:10: warning: type defaults to 'int' in declaration of 'x' [-Wimplicit-int]
    3 |     auto x = 13;
      |
```

when that removed feature finally becomes reported as error after 25 years

```
$ cat a.c
int main()
{
    auto x = 13;
    return 0;
}
$ gcc -std=c99 a.c
```

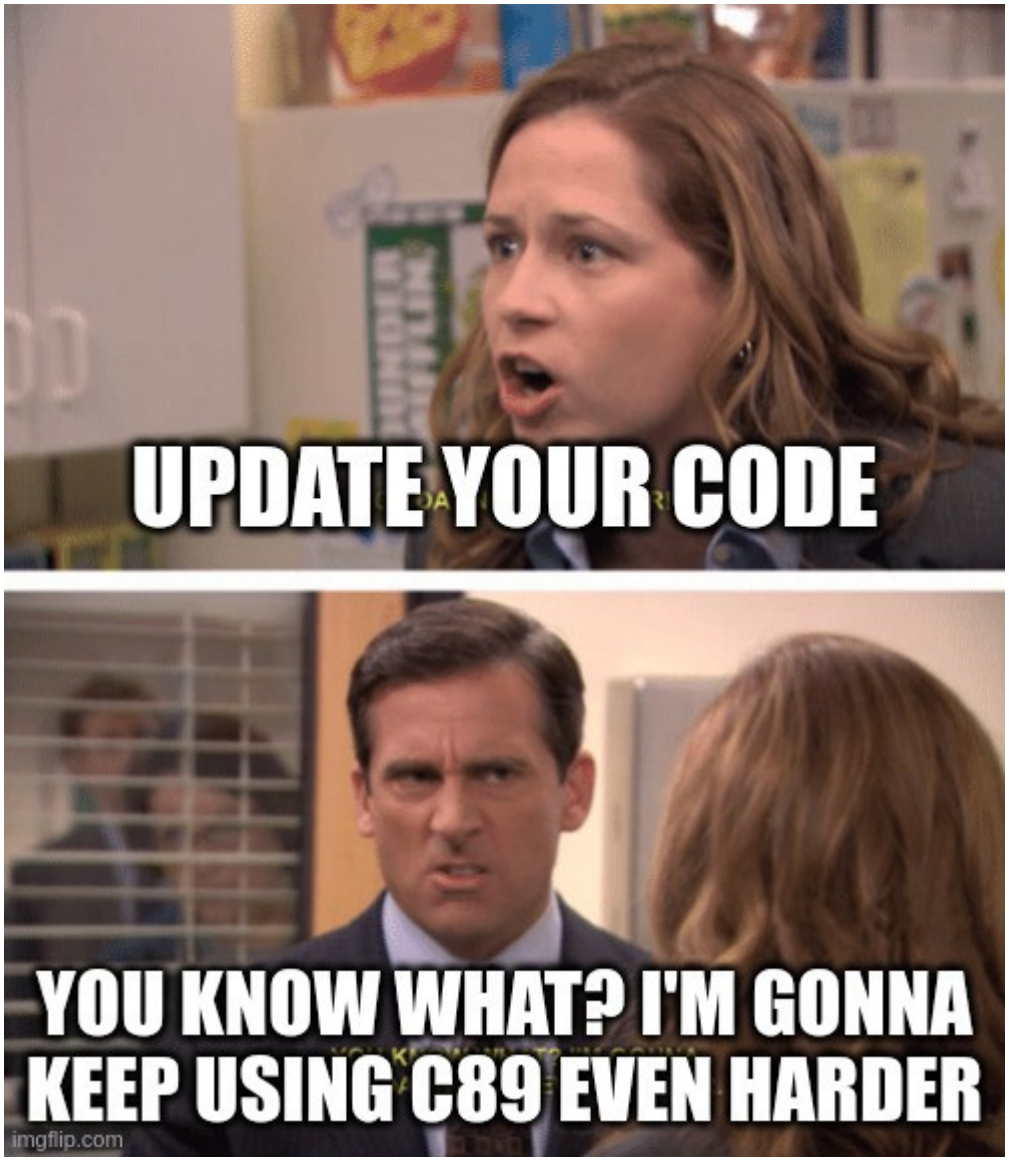
Debian 12	*	success
Debian 13	*	failure?!
Arch Linux	before May 2024	success
	after May 2024	failure?!

```
$ cat a.c
int main()
{
    auto x = 13;
    return 0;
}
$ gcc -std=c99 a.c
```

Should be:

*	*	failure!

Folks like their old C standards



Folks like their old C standards

CURL AND LIBCURL

CONSIDERING C99 FOR CURL

NOVEMBER 17, 2022 DANIEL STENBERG 1 COMMENT

tldr: we stick to C89 for now.

https://daniel.haxx.se/blog/2022/11/17/considering-c99-for-curl/

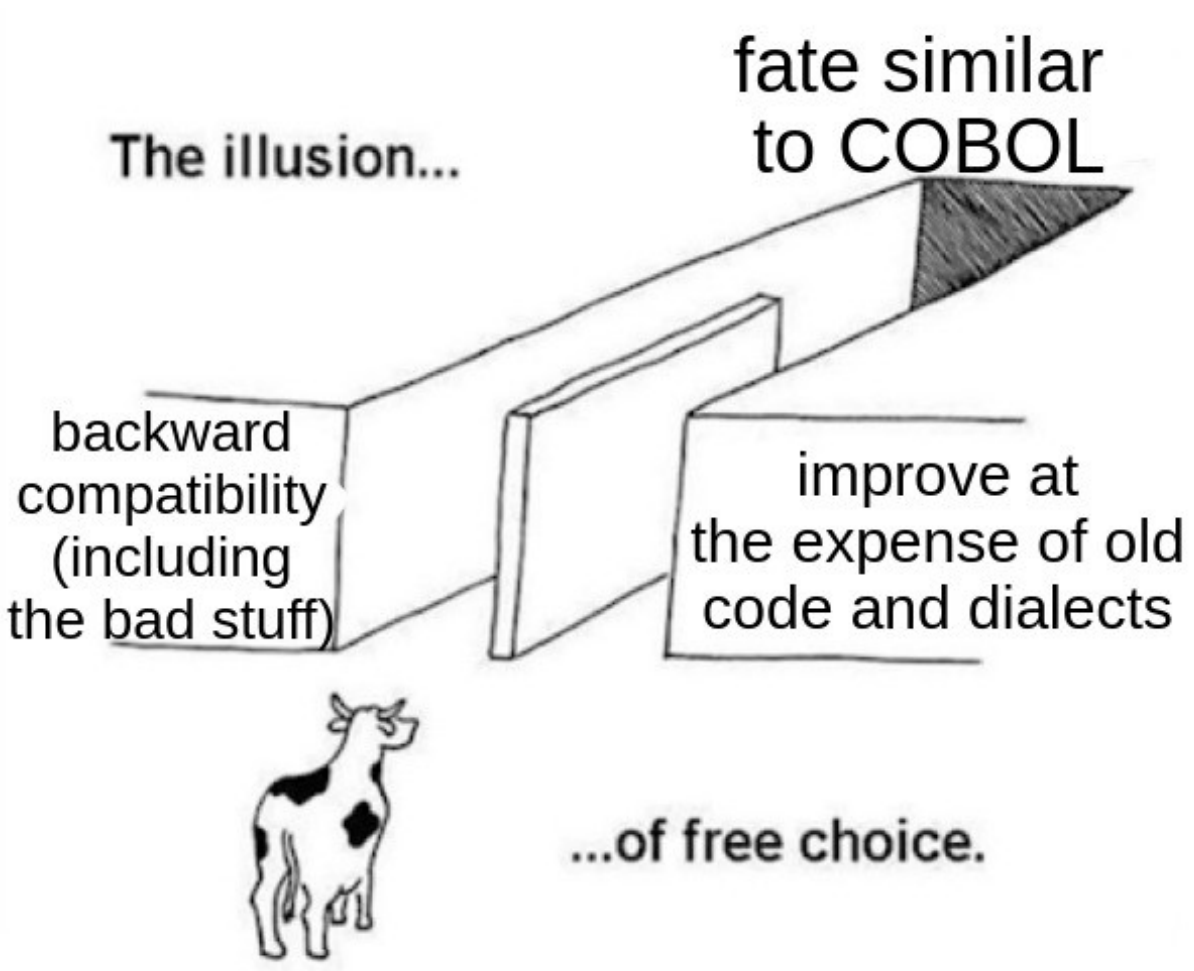
And "if it ain't broke, don't fix it"

- Fixing style violations while working on a real change as a preparatory clean-up step is good, but otherwise avoid useless code churn for the sake of conforming to the style.

  "Once it _is_ in the tree, it's not really worth the patch noise to go and fix it up."
  Cf. https://lore.kernel.org/all/20100126160632.3bdbe172.akpm@linux-foundation.org/

https://git.kernel.org/pub/scm/git/git.git/tree/Documentation/CodingGuidelines

The dilemma C faces



* "slightly" exaggerated for dramatic effect

SOLUTION

`#dialect` directive

a piece of metadata telling the compiler
under which rules the code was written

or: `#version`, `#lang`, `#ver`, `#std`, etc.

What does it entail?

- pre-processor directive `#dialect`
  - propagates to included files
  - does **not** propagate back
  - emits diagnostic on unsupported dialects
    - the standard itself requires only support of its own revision
    - implementation that offer partial support recommended to display warning listing unsupported features or what conflicts it has
- way to query for currently active dialect,
  as well as available ones (e.g. `__has_dialect`)
- attribute for marking objects as dialect “exclusive”
  - + figuring out the best way to handle library features available conditionally by dialect

Optional / further additions / to consideration

- analogous pragma, but which propagates back
- push and pop functionality
- user-defined dialects a la D’s versions?

Rules

For the sake of simplicity the examples across this document use rather loose set of rules for the directive. The final feature will be surely defined more robust*, something akin to:

- Unless specified in other way, if preceded only by comments, `#if` and `#define` directives, the first `#dialect` directive sets base dialect for the translation unit.
  - If no dialect is set explicitly before the first non comment, `#if` or `#define`, then value of dialect is implementation defined.
- No dialect based on higher version of C standard than the one set as the base dialect shall be used within the TU.
- `#dialect` directive shall appear only on file scope.
- Dialect value shall be carried into included files.
- After finishing processing of included file, dialect value from before inclusion shall be restored.

* The final syntax as well ought to provide finer level of control and precision.

The directive **communicates** to compiler what **assumptions** were made **by programmer**.

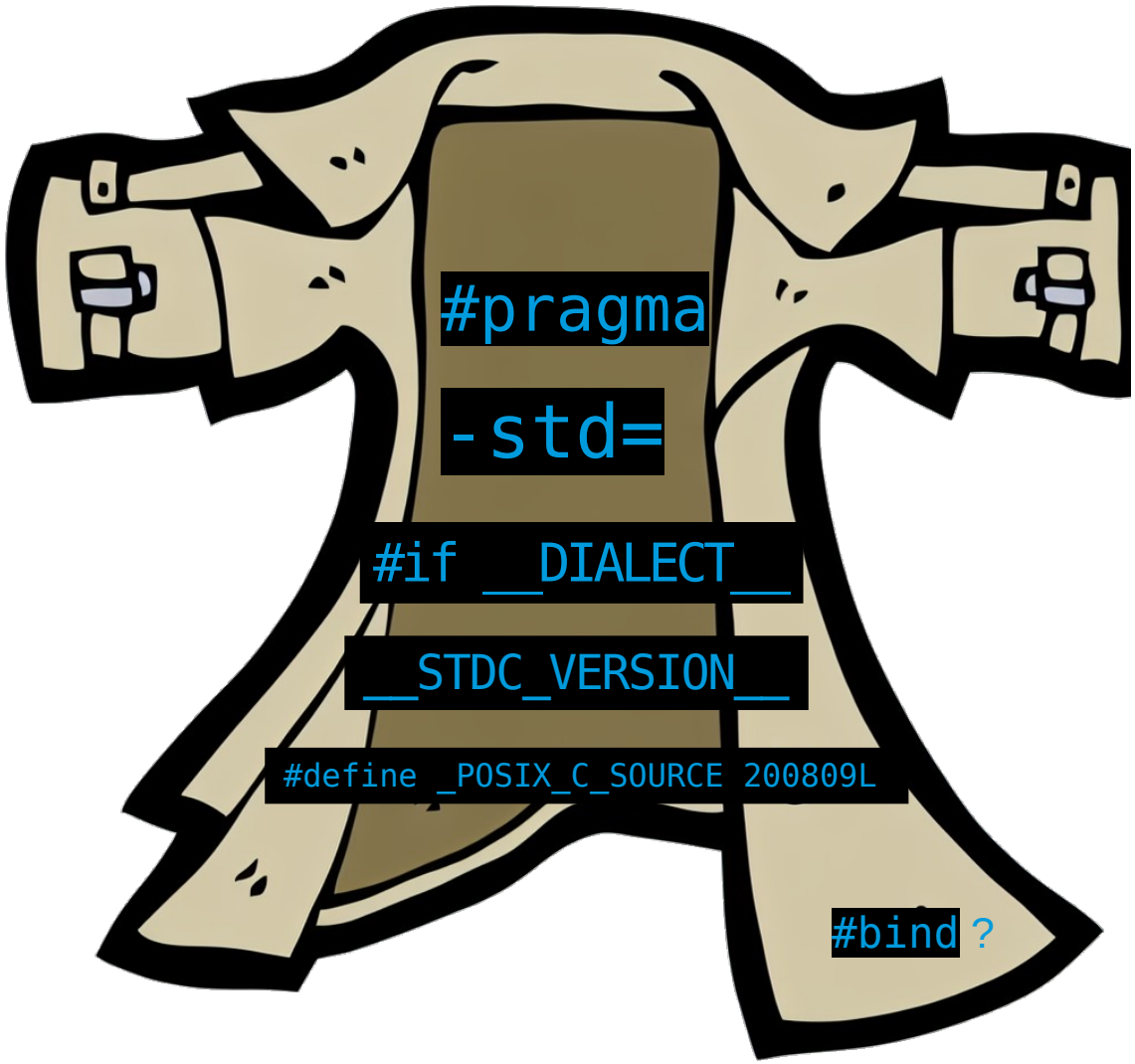
The behaviour is **still the same as before**:
implementation defined
but now explicitly, instead of implicitly.

PRIOR ART

* non-extensive list

Existing C features

`#dialect` =



Perl

```
sub say {
    my ($input) = @_;
    print "$input World!\n";
}
say("Hello");
*old_say = \&say;

use v5.10;  # Perl 5.10 added 'say'

say("Hello");
old_say("Hello");
```

Output:

- Hello World
- Hello
- Hello World

BENEFITS

- What problems does it solve?
- What improvements and fixes it enables?

Disclaimer:
Examples on the following slides aren't meant to be perfectly accurate, but to convey the idea.

(tiny) step towards simpler build systems

(tiny) step towards simpler build systems

Especially for techniques like:

- unitary builds,
- X-Files,
- subtree dependencies,
- etc.

Correct diagnostics for single files

```c
int bar();

void foo()
{
    bool b1 = {};
    _Bool b2 = false;
    int *p = nullptr;

    bar(1);

JUMP:
    int x = {};
    goto LABEL;

    {
LABEL:
    }

    int a = 11'11;
    if (a+x && b1 && b2 && p)
        goto JUMP;
}
```

(default linting of C23 code by **clangd 19.1.6**)

Correct diagnostics for single files

```
int bar();

void foo()
{
    bool b1 = {};
    _Bool b2 = false;
    int *p = nullptr;

    bar(1);

JUMP:
    int x = {};
    goto LABEL;

    {
LABEL:
    }

    int a = 11'11;
    if (a+x && b1 && b2 && p)
        goto JUMP;
}
```

```
/* compile_commands.json */
[
  {
    "command": "cc -std=c23 -c x.c",
    "directory": "/path/to/dir",
    "file": "x.c"
  }
]
```

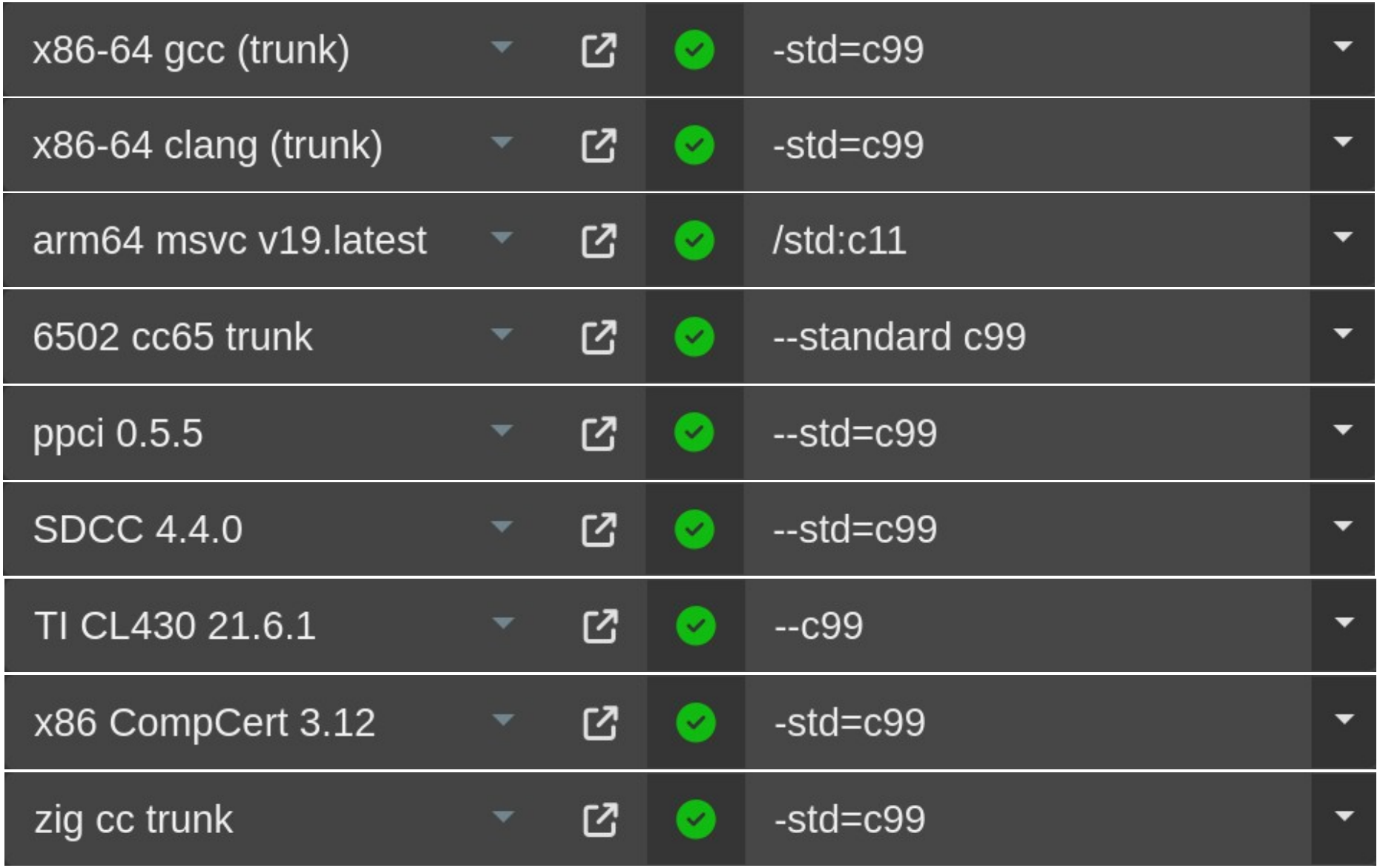
← had to be saved on disk

Manually written,
because build fails!

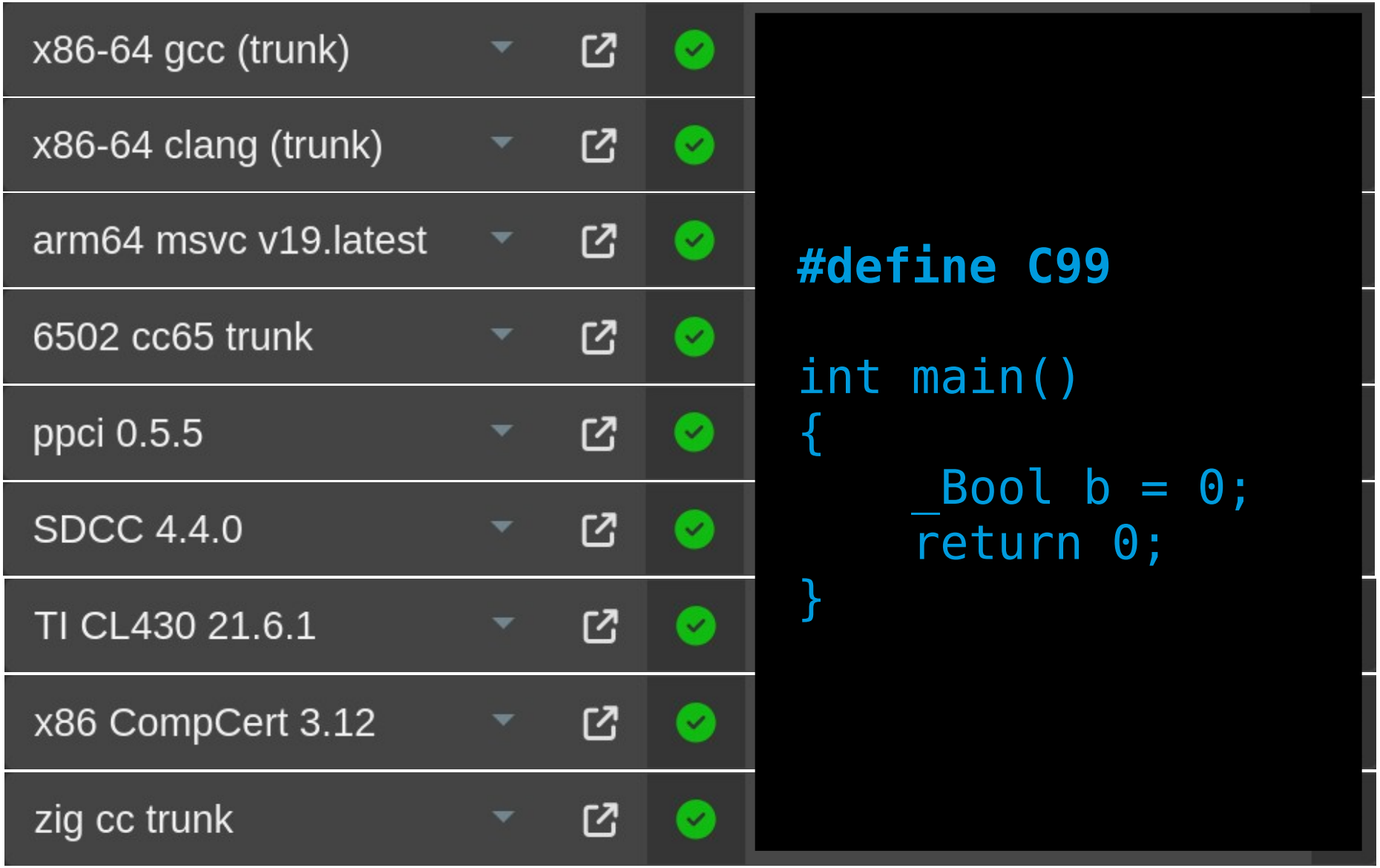
Correct diagnostics for single files

```
#dialect C23

int bar();

void foo()
{
    bool b1 = {};
    _Bool b2 = false;
    int *p = nullptr;

    bar(1);

JUMP:
    int x = {};
    goto LABEL;

    {
LABEL:
    }

    int a = 11'11;
    if (a+x && b1 && b2 && p)
        goto JUMP;
}
```

No need to constantly change flags when testing features across implementations e.g. on Compiler Explorer



No need to constantly change flags when testing features across implementations e.g. on Compiler Explorer



No need for _UglyKeywords

```
#dialect C89
#include <lib_bool.h>
void (*my_bool)(int,int) = bool;

#dialect C23
int main()
{
   bool x = false;
   my_bool(13, 44);
   return 0;
}
```

```
/* lib_bool.h */

void bool(int,int);
```

```
#include "lib_bool.h"

void bool(int a, int b)
{
   int _Bool = a+b;
}
```

https://www.open-std.org/jtc1/sc22/wg14/26816

No need for _UglyKeywords

```
#dialect C23
#include <lib.h>
const auto lib_lengthof = lengthof;


#dialect C2y
void func()
{
   auto n = lengthof (int[12]);
   auto l = lib_lengthof(/* args */);
}
```

Full power of ~~_Optional~~ optional

```
/* old_lib.c */
void foo(int *p)
{
   if (p) {
      *p = 12;
   }
}
```

```
/* new_lib.c */
void bar(int *p)
{
   *p = 12;
}
```

```
#define NULL ((void*)0)

#include <old_lib.h>
#include <new_lib.h>

void func()
{
   foo(NULL); // no warning (good?)
   bar(NULL); // no warning (bad!)
}
```

Full power of optional

```
/* old_lib.c */
void foo(int *p)
{
   if (p) {
      *p = 12;
   }
}
```

```
/* new_lib.c */
void bar(int *p)
{
   *p = 12;
}
```

```
#define NULL ((optional void *)0)

#include <old_lib.h>
#include <new_lib.h>

void func()
{
   foo(NULL); // false positive (bad)
   bar(NULL); // warning (good!)
}
```

Full power of `optional`

```
/* old_lib.c */
void foo(int *p)
{
   if (p) {
      *p = 12;
   }
}
```

```
/* new_lib.c */
void bar(int *p)
{
   *p = 12;
}
```

```
#define NULL ((optional void *)0)

#dialect C23
#include <old_lib.h>

#dialect C2y
#include <new_lib.h>

void func()
{
   foo(NULL); // info note (good)
   bar(NULL); // error (very good!)
}
```

Removal of 0 as null (N3426 alt.1)

0 as null is common not only directly, but also through such idioms:

```
$ cat a.c
int main()
{
   struct { int *ptr; int a; } foo = { 0 };
   return 0;
}

$ gcc -Wzero-as-null-pointer-constant a.c
a.c: In function ‘int main()’:
a.c:3:34: warning: zero as null pointer constant
    3 |     struct { int *ptr; } foo = { 0 };
      |
```

```
a.c:3:43: warning: ISO C forbids empty initializer braces before C23
```

Removal of `0` as null (N3426 alt.1)

```
/* old_lib.h */

#dialect C99

struct Foo {
    char *name;
    int number;
};

static inline void func()
{
    struct Foo x = { 0 };
    int *p = 0;
}
```

```
#dialect C2y
#include <old_lib.h> // fine

int main()
{
    struct Foo y = {};
    struct Foo z = {0}; // err
    int *ptr = 0;       // err
}
```

Avoids repeat of mess like with implicit `int`;
benefits available immediately for new, and refactored code.

Finally `-Wwrite-strings` by default?

```
#include <stdio.h>

#define PRINT_STR_TYPE()                \
    puts(_Generic("",                   \
         const char *: "const char *",  \
               char *:       "char *"   \
    ))

#dialect C99
void foo99()
{
    PRINT_STR_TYPE();
}

#dialect C2y
void foo2y()
{
    PRINT_STR_TYPE();
}

int main()
{
    foo99(); // out: "char *"
    foo2y(); // out: "const char *"
    return 0;
}
```

The `auto` shenanigans

While implicit `int` officially was removed in C99, in practice major implementation**s** didn't follow suit.

Until very recently no error was raised, only warnings without any mentioning of even deprecation.

Therefore, albeit small, there is non 0% chance for snippets like

```
auto x = pow(f,2);
```

to cause subtle, and difficult to detect, yet potentially disastrous changes in behaviour of the code.

Without `#dialect C89`, prints "8" instead of expected "7"!

```
#include <stdio.h>
#include <math.h>

#dialect C89
int idx(float f)
{
    auto x = pow(f,2);
    auto y = pow(f,3);
    return x+y;
}

const char *secret[] = {
    "1", "2", "3", "4", "5",
    "6", "7", "8", "9", "10"
};

#dialect C23
int main()
{
    auto i = idx(1.7);
    puts(secret[i]);
    return 0;
}
```

Non-obtrusive warnings for obsolete octals

```
/* mod.h */
int mod = 0602;
```

```
#dialect C89
#include <mod.h>   // no warning

#dialect C2y

int day   = 01;   // warning
int month = 02;   // warning
int year  = 2025;

int new_mod = 0700; // warning
```

Attributisation of `restrict`

`restrict` keyword has characteristics of attributes and probably ought to be redefined as such, but for ***very*** unlikely scenario of somebody relaying on it being a qualifier:

```
#define foo(X) _Generic((X),   \
       int * restrict: "123",  \
       int *: "456"            \
    )

#dialect Weird-C23
void bar(int * p, int * restrict r)
{
    puts(foo(p)); // 456
    puts(foo(r)); // 123
}

#dialect C23
void baz(int * p, int * restrict r)
{
    foo(r); // warning: ... association 'int *restrict' will never be selected
}

#dialect C2y
void bay(int * restrict r); // warning: using `[[restrict]]` attribute
```

Fixing/improving decay to pointer?

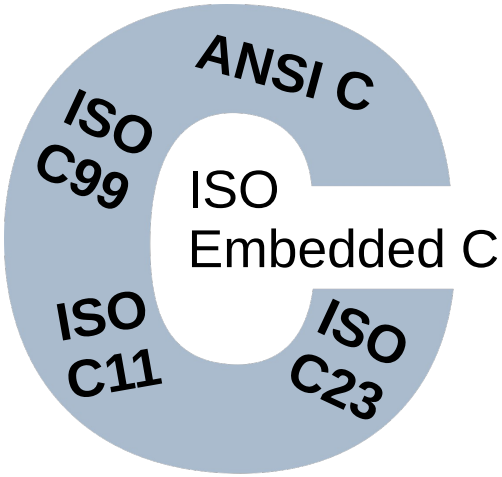
```
#dialect C99
void foo99(int *arr, int n);
void bar99(int arr[], int n);

#dialect C2y
void foo2y(int *ptr, int n);
void bar2y(int arr[], int n);

void foo()
{
    int a[] = { 1, 2, 3 };
    int *p = nullptr;

    foo99(a, lengthof a);  // no error, maybe warning
    bar99(a, lengthof a);
    foo99(p, 0);
    bar99(p, 0);           // no error, maybe warning

    foo2y(a, lengthof a);  // error, or at least a warning
    bar2y(a, lengthof a);
    foo2y(p, 0);
    bar2y(p, 0);           // error, or at least a warning
}
```

Fixing/improving decay to pointer?

Maybe some syntax sugar?

```
#dialect C2y

void foo(size_t n)
{
   int (*arr1)[n] = malloc(sizeof *arr);
   arr[1] = 4; // syntax sugar for `(*arr)[1]`

   int arr2[10];
   &arr2; // type is `int (*)[10]` instead of `int*`
}
```

Better interoperability with dialects, forks, and other languages



eC

C*

cc65

Open Cilk

C+-

Fil-C

Objective-C

TrapC

CYCLONE

CompCert

OpenCL C

Checked C

Many languages are capable of digesting C code more or less "directly"

Better interoperability with dialects, forks, and other languages

Got a bunch of C code that uses *operator* as a variable name. Since this is a C++ keyword, it will be tough to import this into C++ code.

I told them to rename all their *operator* vars to *var_operator* and they were like "yeah no."

~ u/apple_lie, 2025-01-10

https://www.reddit.com/r/C_Programming/comments/1hy63n2/

Make `#dialect` work like people **think** `extern "C"` works?

Possibility of an extension in C++:

```
/* foo.h */

#dialect C2y

// not needed: #if __cplusplus \ extern "C"

static void foo(size_t n, int arr[n]) // VM type!
{
   char *operator = malloc(89); // no cast
   printf("%zu\n", lengthof arr);
}
```

```
#dialect GNU++3a
#include <foo.h>  // actually compiles as C

auto example::func() -> void
{
   foo();
}
```

Standardizing popular extensions with different syntax

GNU C

```
asm (
 "idivl  %[divsrc]"
 : "=a" (quotient), "=d" (rem)
 : "d" (hi), "a" (lo),
 [divsrc] "rm" (divisor)
 :
);
```

ISO C

```
asm ??
```

Microsoft style

(in Intel's "classic" compiler?)

```
asm {
    mov   edx, hi;
    mov   eax, lo;
    idiv   divisor
    mov   quotient, eax
    mov   tmp, edx;
}
```

Code examples from:
https://stackoverflow.com/a/35959859

Standardizing popular extensions with different syntax

GNU C

```
/* div64.h.c */
#dialect GNU99

int div64(int lo, int hi,
          int *premainder,
          int divisor)
{
    int quotient, rem;
    asm ("idivl  %[divsrc]"
         : "=a" (quotient), "=d" (rem)
         : "d" (hi), "a" (lo),
         [divsrc] "rm" (divisor)
         :
    );
    *premainder = rem;
    return quotient;
}
```

ISO C

```
#include "div64.h.c"

#dialect C2y
void foo()
{
    asm "
        some
        other
        syntax
    ";
}
```

Code examples from:
https://stackoverflow.com/a/35959859

Avoiding conflicts with extensions

What if C adds trap mechanism but not like e.g. from TrapC (N3423) dialect?

```
#dialect C3a

int func(size_t n)
{
    trap VLA_OVERFLOW { return 1; }
    int arr[n];
    // work on arr...
    return 0;
}
```

(similar to trap from POSIX shell)

Avoiding conflicts with ambiguous behaviours defined by implementation

For example N3203 standardizes order of expression evaluation.
While unlikely, what if some niche implementation had already defined it, but in reverse?

```
int G = 0;
int f() { return ++G; }
int g() { return (G *= 3); }

#dialect C2y

int s(int a, int b)
{
    return a + b;
}

int main()
{
    return s(f(), g());  // weird-cc notices these can have side effects
    /* warning:
     *  order of evaluation is left to right in C2y;
     *  weird-cc for previous standard used right to left
     */
}
```

Piecemeal updates of source code

This particular mechanism for the directive as shown in examples before allows for gradual update of single file to newer language edition.

```
#dialect C2y

void foo(size_t n; int arr[n], size_t n);
void bar(optional int *p) { if (p) *p = 12; }

/* 1000 lines of updated code */
// PR here (to not overwhelm reviewers)

#dialect C11

void baz(int arr[], size_t n);
void fez(int *p) { if (p) *p = 6; }

/* 5280 more lines of code to update */
```

Whether it's good or bad thing is up to discussion at later date.

In case of only one #dialect per file, the alternative approach would be to split the file and just include one into another (with downside of disturbing valuable commit history).

List of revisions' nicknames in Annex J

Having nicknames "formally" in a form of a clause like § *J.7 Common dialects* could help with such issues:

- Wikipedia article on C17 used to have:

  C18 (previously known as C17) is the informal name for ISO/IEC 9899:2018, (…) It replaced C11 (…)

- Certain guideline also has *«been referring to C17 as "C18" in their documentation and plan to refer to C23 as either "C24" or "C25" depending on the publication date. This has the unfortunate effect of creating confusion about which version of the language is being referenced and other problems.»*

and **more**

QUESTIONS?

Questions?

- Why “dialect”?
  - Feels like best umbrella term that would also encompass forks.
- Won’t this fragment the ecosystem?
  - GCC 13 treats implicit `int` in C99 differently that the same compiler a version later for the same revision of the standard…
  - The goal is i.a. to spare implementations from having to split behaviour from the spec.
- Won’t this force implementers to support old standards forever?
  - Could be, but the relatively small sets of changes between revisions, and short list of breaking changes, combined with pre-existing expectations, already do provide enough incentive anyway.
  - Other languages receive praise for declaring support to their older editions.
- Won’t this cause rise of ABI breaks?
  - The precautions should remain relatively unchanged from current status quo.
- Why not setting pragmas/only on demand?
  - Fine as ad hoc solution, but isn’t future proof.
- Won’t this cause developers leaving old code to “rot”?
  - That happens already, but often instead of being contained to only old parts of codebase, the “rot” drags down also the new code. Static analysers, guidelines, requirement-makers should still insist on updating to latest (presumably) safer and better editions.

POLLS

- Does WG14 want something like `#dialect` directive in C2y?

Thank you!

Acknowledgments:

- Eskil Steenberg (N3176), Ori Bernstein
- Bartosz Zielonka, and MTP team
- All WG14 participants in discussion on mailing list
- SWC audience