SDCC

ISO C Standard Compliance

Benedikt Freisen

2025-04-11

Structure of this talk

This talk will outline the status quo and objectives surrounding C standard compliance in SDCC, structured as follows:

- Motivation: Why ISO C?
- Recent Achievements
- Work(s) in Progress
- Challenges
- Modern C — Examples
- Future Construction Sites
- Time for Questions / Feedback / Live Demo

Motivation: Why ISO C?

The C programming language was originally standardized by ANSI in 1989, all subsequent versions by ISO

- Features: Language extensions for new use cases
- Compatibility: Prevention of diverging dialects

Therefore, we want a Standard C compiler! (ISO C90 - C2Y)
Obvious challenge: Standard C is constantly evolving

Recent Achievements (pre 4.5.0)

Six out of the seven new features in 4.5.0 were related to current and future standard compliance:

- Full atomic_flag support for msc51 and ds390 ports
- (Experimental f8 port)
- ISO C2y case range expressions
- ISO C2y _Generic selection expression with a type operand
- K&R-style function syntax (preliminarily with the semantics of non-K&R ISO-style functions)
- ISO C23 enums with user-specified underlying type
- struct / union in initializers

Recent Achievements (post 4.5.0)

Five out of the six features added since 4.5.0 relate to ISO C, one to GNU C:

- C2y _Countof operator
- C2y octal
- C2y if-declaration
- Conditional operator with omitted second operand (GNU extension)
- C99 compound literals
- C23 compound literals with storage class specifiers

Work(s) in Progress

Various ISO C features have already been worked on, but are not yet complete, e.g.:

- C23 constexpr, i.e. named compile-time constant expressions
- C23 qualifier-preserving library functions
- C90 flat initializer lists for nested arrays
- C23 checked integer arithmetic for 64 bit types

Challenges

There are obstacles preventing the speedy implementation of the aforementioned features:

- Qualifier-preserving wrapper macros rely on _Generic and ?: specifics that are inaccurately implemented
- Flat initilizer lists can contain (top level) element designators, but we have a bottom-up parser
- constexpr must not trip over identifier shadowing
- 64 bit checked integer arithmetic must make do with 64 bit intermediate values

Challenges: Flat Array Initializer Lists

ANSI/ISO C inherits the K&R C bug/feature that initializer lists may ignore an array type's structure. This is valid:

```c
int array[2][2] = { 1, 2, 3 };
```

However, ISO C99 added the ability to designate specific array members:

```c
int array[2][2] = { 1, [2] = 2, 3 };
```

Mixing both means trouble! (At least for compiler devs)

Modern C — Examples: _Generic

Generics (since C11) allow selecting expressions based on the type of a controlling expression:

```
_Generic(i, default : 0, int : 1, long : 2)
```

C2y will add the ability to select based on a type itself, which, combined with C23's `typeof` permits selection based on the *qualified* type:

```
_Generic(typeof(i), int : 0, const int : 1, default : 2)
```

Modern C — Examples: `typeof` with function pointers

C23's `typeof` permits a more intuitive spelling of function pointers:

```
void (*signal(int sig, void (*func)(int)))(int);
```

becomes

```
typeof(void (int)) *signal(int sig, typeof(void (int)) *func);
```

But SDCC does not like this syntax, yet. :-/

Modern C — Examples: case ranges

C2y will *finally* standardize GNU C's case ranges:

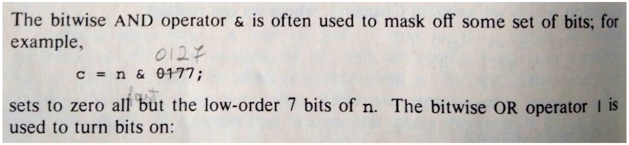
```
switch (a) {
    case 0 ... 5: foo(); break;
    case 6 ... 10: bar(); break;
    default: baz(); break;
}
```

The implementation in SDCC is based on a user-contributed patch. Those are welcome![1]

[1] Particularly when a feature has been standardized

Modern C — Examples: Octal Literals

Octal literals in C are notoriously unintuitive, as demonstrated by this pencil "correction" in a library copy of the book "The C Programming Language":



The bitwise AND operator & is often used to mask off some set of bits; for example,

```
           0127
c = n & 0177;
```

sets to zero all last but the low-order 7 bits of n. The bitwise OR operator | is used to turn bits on:

C2y will allow the more obviously prefixed spelling `0o177` known from languages like Rust.
It will also allow escape sequences like "\o{177}".

Future Construction Sites

Besides the points that are already being worked on, the following ones remain:

- Changes to lexer and parser for better C23 attribute support
- Proper IEEE 754 floating point types and library
- Whatever the committee decides to put in C2y, e.g.
    - Lambda expressions (subset of what C++ has)
    - Deferred blocks (executed at the end of the containing block)
    - Named loops (allowing e.g. `continue outer;`)

All of the above will presumably come with their own unique challenges.

Time for Questions / Feedback / Live Demo

- Any questions?
- Do you have feedback or suggestions?
- Would you like to see details?