n2530 - Allow compound literals of static lifetime inside body functions

Proposal for the upcoming C2X standard
Submitter: Xavier Del Campo Romero
Submission date: 2020-06-04

Summary

The current standard ISO/IEC 9899 6.5.2.5 (5) states:
"If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block."
The following C11-compliant example provided below makes use of various C99 features (compound literals, designated initializers and variadic macros) and allows defining an array of instances containing arrays of arbitrary size at compile-time:

```c
#include <stddef.h>
#include <stdio.h>

typedef const struct {
    size_t len;
    const char *buf;
} transfer;

#define TRANSFER(...) \
    { \
        .len = sizeof (const char[]){__VA_ARGS__} \
            / sizeof *(const char[]){__VA_ARGS__}, \
        .buf = (const char[]){__VA_ARGS__} \
    }

static transfer tr[] = {
    /* Using the optional convenience macro. */
    TRANSFER(1, 2, 3, 4, 5, 6, 7, 8),

    /* Or without using the optional convenience macro. */
    {.len = 5, .buf = (const char[]){1, 2, 3, 4, 5}},
    {.len = 3, .buf = (const char[]){1, 2, 3}}
};

int main(const int argc, const char *argv[]) {
    for (size_t i = 0; i < sizeof tr / sizeof *tr; ++i) {
        transfer *const t = &tr[i];

        for (size_t j = 0; j < t->len; ++j) {
            printf("operating on tr[%zu].buf[%zu] (0x%02X)\n",
            i, j, tr[i].buf[j]);
        }
    }

    return 0;
}
```

The problem

As shown, this code snippet will execute specific actions for each element in the arrays, while considering how many elements have been allocated for each instance. Although it works as expected on any C11-compliant implementation, transfer and tr are defined at a file scope, but are only used by main.

However, any attempt to move *transfer* and *tr* inside the body of *main* will trigger a compile-time error since compound literals are not static inside a function body. For example, GCC 7.5.0 triggers the following error on the modified example below:

```
$ gcc ex1.c
ex1.c: In function ‘main’:
ex1.c:15:16: error: initializer element is not constant
        .buf = (const char[]){__VA_ARGS__} \
               ^
ex1.c:22:9: note: in expansion of macro ‘TRANSFER’
        TRANSFER(1, 2, 3, 4, 5, 6, 7, 8),
        ^~~~~~~~
ex1.c:15:16: note: (near initialization for ‘tr[0].buf’)
        .buf = (const char[]){__VA_ARGS__} \
               ^
ex1.c:22:9: note: in expansion of macro ‘TRANSFER’
        TRANSFER(1, 2, 3, 4, 5, 6, 7, 8),
```

```c
#include <stddef.h>
#include <stdio.h>

int main(const int argc, const char *argv[]) {

    typedef const struct {
        size_t len;
        const char *buf;
    } transfer;

#define TRANSFER(...) \
    { \
        .len = sizeof (const char[]){__VA_ARGS__} \
            / sizeof *(const char[]){__VA_ARGS__}, \
        .buf = (const char[]){__VA_ARGS__} \
    }

    /* Compiler error generated due to compound literal having
    automatic storage instead of static. */
    static transfer tr[] = {
        /* Using the optional convenience macro. */
        TRANSFER(1, 2, 3, 4, 5, 6, 7, 8),

        /* Or without using the optional convenience macro. */
        {.len = 5, .buf = (const char[]){1, 2, 3, 4, 5}},
        {.len = 3, .buf = (const char[]){1, 2, 3}}
    };
```

```
    for (size_t i = 0; i < sizeof tr / sizeof *tr; ++i) {
        transfer *const t = &tr[i];

        for (size_t j = 0; j < t->len; ++j) {
            printf("operating on tr[%zu].buf[%zu] (0x%02X)\n",
            i, j, tr[i].buf[j]);
        }
    }

    return 0;
}
```

Proposal

In this proposal, it is suggested to allow developers to qualify compound literals as *static* so compound literals with static lifetime can still be used inside the body of a function.

A modified version of the example above introduces the static keyword to compound literals, so both *transfer* and *tr* can be moved inside *main*:

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

int main(const int argc, const char *argv[]) {

    typedef const struct {
        size_t len;
        const char *buf;
    } transfer;

#define TRANSFER(...) \
    { \
        .len = sizeof (static const char[]){__VA_ARGS__} \
            / sizeof *(static const char[]){__VA_ARGS__}, \
        .buf = (static const char[]){__VA_ARGS__} \
    }

    /* Compiler error generated due to compound literal having
    automatic storage instead of static. */
    static transfer tr[] = {
        /* Using the optional convenience macro. */
        TRANSFER(1, 2, 3, 4, 5, 6, 7, 8),

        /* Or without using the optional convenience macro. */
        {.len = 5, .buf = (static const char[]){1, 2, 3, 4, 5}},
        {.len = 3, .buf = (static const char[]){1, 2, 3}}
    };

    for (size_t i = 0; i < sizeof tr / sizeof *tr; ++i) {
        transfer *const t = &tr[i];

        for (size_t j = 0; j < t->len; ++j) {
```

```
            printf("operating on tr[%zu].buf[%zu] (0x%02X)\n",
            i, j, tr[i].buf[j]);
        }
    }

    return 0;
}
```

Please take into account the static compound literals used to determine array length (via struct member *len*) in macro *TRANSFER* are not in fact stored into memory, but are optimized away by the compiler since they cannot be accessed from anywhere but from the *sizeof* operators.

Reason behind this proposal

Encouraging developers to reduce the scope of any symbol to its absolute minimum reduces the risk of accessing and/or modifying data accidentally, improves modularity by encapsulation and enhances readability. Allowing static compound literals to be defined inside the body of a function provides all these benefits without introducing any breaking changes to existing code, and should not imply great difficulty for currently available implementations.

Other examples

The example that has been used so far has proven useful for tasks such as TFT LCD initialization, where a register is accessed via microcontroller communication peripherals (e.g.: SPI, I2S, I2C, etc.) and an arbitrary number of parameters must be sent afterwards, where the number of parameters is tied to the accessed register. Whereas other implementations use a worst-case scenario and set the same array length to all struct instances, this implementation proves to be more efficient by only allocating the strictly amount of bytes needed per instance.

Below there is another minimalistic example of static compound literals (this time, non-*const*) that allow for an implementation of a round-robin scheduler for a small microcontroller where tasks are defined by a callback and how often, in seconds, they must be executed. Another struct member, *rt*, holds mutable parameters, which are grouped into another struct named *process_rt*:

```
typedef const struct
{
    void (*callback)(void);
    int seconds;
    struct process_rt *rt;
} process_cfg;

/* Runtime information. */
struct process_rt
{
    process_cfg *next;
    /* Other runtime parameters. */
};

void init(void)
{
    static process_cfg cfg =
    {
```

```
        .callback = second_task,
        .seconds = 1,
        /* Allocate statically needed runtime parameters. */
        .rt = &(static struct process_rt){}
    };

    scheduler_add(&cfg);
}
```

By using the static qualifier, it allows the *process_cfg* instance to hold a *process_rt* instance that cannot be accessed outside it. A C11-conforming program would have to allocate the *process_rt* instance separately and refer to it via a pointer, as shown below:

```
typedef const struct
{
    void (*callback)(void);
    int seconds;
    struct process_rt *rt;
} process_cfg;

/* Runtime information. */
struct process_rt
{
    process_cfg *next;
    /* Other runtime parameters. */
};

void init(void)
{
    static struct process_rt rt;

    static process_cfg cfg =
    {
        .callback = second_task,
        .seconds = 1,
        .rt = &rt
    };

    scheduler_add(&cfg);
}
```

While not a severe drawback on this minimalistic example, it can be worsened if multiple *process_cfg* instances were defined on the same function, affecting readability and increasing the risk for human errors.

Other considerations

Outside the body of a function, the *static* qualifier would have no effect on compound literals as they already have static lifetime according to the current standard, similarly to how *extern* does not affect function declarations as functions have global linkage by default.