Allowing access to enclosing object using `offsetof` (Slides for P3407R1)

Brian Bi
Document number: P3601R0
Presented to EWG on June 17, 2025

Intrusive data structures in C

```c
struct ListNode {
    struct ListNode* prev;
    struct ListNode* next;
};

typedef struct {
    int data;
    struct ListNode node;
} Foo;

Foo* next_foo(Foo* foo) {
    struct ListNode* next_node = foo->node;
    return (Foo*)((char*)next_node - offsetof(Foo, node));  // <----- UB in C++
}
```

Proposal: make this do the right thing in C++

- There is no better way to write the code in C.
- The behavior used to be well-defined before C++17.
- We should restore compatibility with C.
- **No implementations need to change**. They do the right thing already.

Problem 1: Pointer arithmetic within objects

We don’t define the meaning of this subexpression:

```
(char*)next_node - offsetof(Foo, node)
```

`next_node` doesn’t point into an array of `char`.

Pointer arithmetic within object representations is the subject of P1839R7. But that paper doesn’t address “escaping” from a subobject into the enclosing object.

Problem 2: Reachability

```
struct S {
    int x;
    int y;
};

void modify_S_x(int* py);

int f() {
    S s {};
    modify_S_x(&s.y);
    return s.x * s.x;  // Compiler can optimize this to return 0;
}
```

Problem: `s.y` is not *reachable from* a pointer to `s.x`

But no C++ compiler is ever going to do that!

- C++ compilers need to consume C code and link with C translation units.
- C code *can* access the enclosing struct given a pointer to a subobject.
- Implementing the optimization would silently break such code, giving it the unbounded behavior of UB. This is presumably why no compilers do it.
- We should just standardize existing practice!

But there are some subtleties

What if the member we start from already has type `char` or array of `char`?

```
struct S2 {
    int data;
    char buf[100];
};
int get_data(char* p) {
    return ((struct S2*)(p - offsetof(S2, buf)))->data;  // out of bounds pointer arithmetic
}
void f5() {
    S2 s;
    // ...
    get_data(s->buf);
    // ...
}
```

But there are some subtleties

What if the member we start from already has type `char` or array of `char`?

- Can you just subtract from it to get to the start of the enclosing object?
- Or do you need to cast the pointer to its own type, `char*`, first?
- Or do you need to cast to a different type, `unsigned char*`?

Compilers already let you do it in all three cases… but sanitizers might have a different opinion.

I propose the last option. We can relax it later if we have to. More analysis is in the paper itself.

But there are some subtleties

This cast to `char*` already has a different meaning:

```
struct S3 {
    char a;
    int  b;
};
struct S4 {
    char      c;
    struct S3 d;
};

struct S4* get_s4(struct S3* s3) {
    // The inner cast actually produces a pointer to d.a
    return (struct S4*)((char*)s3 - offsetof(S4, d));
}
```

Idea: use “angelic nondeterminism”: you get whichever pointer gives you well defined behavior.

Wording strategy

- In P1839R7:
  - Each subobject has its own object representation array (array of `unsigned char`)
  - You can't escape from a subobject's object representation array to that of the enclosing object
  - *Except* when the subobject is the first member of a standard-layout struct (current reachability rule)
- P3407R1 would go further than P1839R7, in order to enable access to the enclosing object:
  - Every byte of a complete object is reachable from a pointer to *any* part of the complete object
  - Each *complete* object has an object representation array
  - Casting to `unsigned char*` from a pointer to a subobject just puts you somewhere in the object representation of the complete object

Future direction: opt in to dangerous optimizations

- "*I want innocuous-looking code to have UB so that the compiler can make other code go faster*" is bad for safety.
- We should make it easy for beginners to write correct code, and give experts the tools with sharp edges to introduce UB, like `[[assume]]`.
- Perhaps there should be an *opt-in* mechanism to tell the compiler that a pointer to a subobject cannot reach any other members of the enclosing object?
- `restrict` provides a way to do this in ISO C. Appendix A of the paper outlines a possible alternative facility that might pose less specification difficulty in C++, and could be added to C as well. The idea is based on the provenance model of CHERI and Rust: a pointer value remembers the range of bytes it is allowed to reach. An expert has to explicitly narrow that range to enable optimizations.