C++17 creates a practical use of the backward array index operator

devblogs.microsoft.com/oldnewthing/20230403-00

April 3, 2023

Raymond Chen

It is well-known that if `a` is a pointer or array and `i` is an integer, then `a[i]` and `i[a]` are equivalent in C and C++, resulting in hilarity like

```
void haha()
{
    int a[5];
    for (i = 0; i < 5; i++) {
        i[a] = 42;
    }
}
```

There is very little practical use for this equivalency, aside from pranking people.[1]

And then C++17 happened.

One of the changes to the core language in C++17 was stronger order of evaluation rules, formally known as *sequencing*. We previously encountered this when studying a crash that seemed to be on a `std::move` operation.

One of the operations that received a defined order of evaluation is the subscript operator. Starting in C++17, `a[b]` always evaluates `a` before evaluating `b`.

```
int* p;
int index();

auto test()
{
    return p[index()];
}

// Compiled as C++14

    sub     rsp, 40
    call    index       ; call index first
    movsxd  rcx, rax
    mov     rax, p      ; then fetch p
    mov     eax, [rax + rcx * 4]
    add     rsp, 40
    ret

// Compiled as c++17

    push    rbx
    sub     rsp, 32
    mov     rbx, p      ; fetch p first
    call    index       ; then call index
    movsxd  rcx, rax
    mov     eax, [rbx + rcx * 4]
    add     rsp, 32
    pop     rbx
    ret
```

Therefore, if your evaluation of the index may have a side effect on the evaluation of the pointer, you can flip the order to force the index to be calculated first.

```
auto test()
{
    return index()[p];
}
```

Astound your friends! Confuse your enemies!

Bonus chatter: Though I wouldn't rely on this yet. clang implements this correctly, but msvc (v19) and gcc (v13) get the order wrong and still load `p` before calling `index`. (By comparison, icc also gets the order wrong, but the other way: It always loads `p` *last*.)

[1] Another practical use is to bypass any possible overloading of the `[]` operator, as noted in Chapter 14 of *Imperfect C++*:

```
#define ARRAYSIZE(a) (sizeof(a) / sizeof(0[a]))
```

By flipping the order in `0[a]`, this bypasses any possible `a[]` overloaded.

```
std::vector<int> v(5);
int size = ARRAYSIZE(v); // compiler error
```

However, it isn't foolproof. You just need to create a more clever fool: If `v` is a pointer or an object convertible to a pointer, then that pointer will happily go inside the `0[...]`.

```
struct Funny
{
    operator int*() { return oops; }
    int oops[5];
    int extra;
};

Funny f;
int size1 = ARRAYSIZE(f); // oops: 6

int* p = f;
int size2 = ARRAYSIZE(p); // oops: 1
```

Fortunately, you don't need any macro tricks. You can let C++ `constexpr` functions do the work for you:

```
template<typename T, std::size_t N>
constexpr std::size_t array_size(T(&)[N]) { return N; }
```