Don't use unions or pointer casts for type punning

Pieter P

What is type punning?

According to Wikipedia, *"type punning is any programming technique that subverts or circumvents the type system of a programming language in order to achieve an effect that would be difficult or impossible to achieve within the bounds of the formal language"*.
A classic example is the Quake III fast inverse square root function, where the bits of an IEEE 754 floating-point number are interpreted as a 32-bit integer:

```
float y = number;
long  i = *(long *) &y;                 // evil floating point bit level hacking
i       = 0x5f3759df - (i >> 1);        // what the fuck?
y       = *(float *) &i;
```

Other uses include serialization and deserialization, where floating-point numbers or other types are converted to and from arrays of bytes to be transmitted over a network or stored to a file.

In the previous example, an invalid C-style pointer cast was used to carry out the type punning. Another commonly used but equally incorrect method makes use of (or abuses) a union:

```
union {
    float x;
    std::byte bytes[sizeof(x)];
} u;
u.x = 12.34f;
write_to_file(u.bytes, sizeof(u.bytes)); // Error: Undefined Behavior
```

Why can't I use a union for type punning?

You cannot use a union for type punning because you are not allowed to first write to one member of the union, and then read from a different one. [cppreference:union]

Specifically, in the second example above, writing to `u.f` makes it the active member, starting its lifetime. [class.union.general] At most one member can be active at any given time.
Reading from the inactive member `u.bytes` is then not allowed, because its lifetime never began, and reading an object before the beginning of its lifetime invokes Undefined Behavior. [basic.life]

Note: In C, the situation is different, C99 and later standards explicitly allow type punning using unions. [C11: footnote 95]

Why can't I use a pointer or reference cast for type punning?

The C-style casts in the first example are equivalent to `reinterpret_cast` expressions. The rules for such casts are quite complicated, see e.g. cppreference: reinterpret_cast. The casts required for type punning fall under items 5 and 6 on that web page, and these casts are only allowed when the type aliasing rules (sometimes called the *strict aliasing rule*) are satisfied. However, the whole point of type punning is that these type aliasing rules are not fulfilled.
In the first example, `float` and `long` are not *similar* types [conv.qual], so the aliasing rule is violated, and dereferencing the pointer resulting from the cast invokes Undefined Behavior.

One important exception to the strict aliasing rule are the character types `(unsigned) char` and `std::byte`: you can inspect the object representation of any object as an array of bytes through pointers to these character types.

What to use instead

`std::bit_cast`

From C++20 onwards, you can use the `std::bit_cast` function from the `<bit>` header. [cppreference:bit_cast] It performs the type punning in a safe way, additionally checking that both types have the same size, and that they are *TriviallyCopyable*.

The fast inverse square root example can be fixed as follows:

```
float y = number;
auto  i = std::bit_cast<uint32_t>(y);   // evil floating point bit level hacking
i       = 0x5f3759df - (i >> 1);        // what the fuck?
y       = std::bit_cast<float>(i);
```

`std::memcpy`

Before C++20, the only valid way to perform type punning was to use the `memcpy` function, copying the object representation from an object of one type to another object of a different type.

It should be noted that this does not mean that an actual call to the C library function `memcpy` will be emitted. Compiler developers are aware of this use of `memcpy`, and completely optimize it out for type punning use cases, you won't see a call to `memcpy` even with optimizations disabled (`-O0`).

Another valid version of the inverse square root code could be:

```
float y = number;
uint32_t i;
static_assert(sizeof(i) == sizeof(y), "error: different sizes");
std::memcpy(&i, &y, sizeof(i));         // evil floating point bit level hacking
i = 0x5f3759df - (i >> 1);              // what the fuck?
std::memcpy(&y, &i, sizeof(y));
```

The `memcpy` function can also be used to fix the second example of converting a float to the bytes it consists of:

```
float x = 12.34f;
uint8_t bytes[sizeof(x)];
std::memcpy(bytes, &x, sizeof(x));
```

Cast to a character array

In many cases related to serialization, using `memcpy` or `bit_cast` is unnecessary, thanks to the exception to the type aliasing rules for character types.

For example, to write the bytes representing a `float` to a file, one could use a cast to a pointer to a character type:

```
float x = 12.34f;
write_to_file(reinterpret_cast<const std::byte *>(&x), sizeof(x)); // Ok
```

Keep in mind though that this is an exception to the rule. It would not be valid to do the same in reverse, for example:

```
uint8_t bytes[] {0xA4, 0x70, 0x45, 0x41};
float x = *reinterpret_cast<float *>(bytes); // Error: Undefined Behavior
```

C++ Core Guidelines

If you're unconvinced by what was presented on this page, you might want to have a look at what the official C++ Core Guidelines have to say about type punning:

- ES.48: Avoid casts
- C.183: Don’t use a `union` for type punning