Chapter 36: Aliasing and effective type

Section 36.1: Effective type

The *effective type* of a data object is the last type information that was associated with it, if any.

```
// a normal variable, effective type uint32_t, and this type never changes
uint32_t a = 0.0;

// effective type of *pa is uint32_t, too, simply
// because *pa is the object a
uint32_t* pa = &a;

// the object pointed to by q has no effective type, yet
void* q = malloc(sizeof uint32_t);
// the object pointed to by q still has no effective type,
// because nobody has written to it
uint32_t* qb = q;
// *qb now has effective type uint32_t because a uint32_t value was written
*qb = 37;

// the object pointed to by r has no effective type, yet, although
// it is initialized
void* r = calloc(1, sizeof uint32_t);
// the object pointed to by r still has no effective type,
// because nobody has written to or read from it
uint32_t* rc = r;
// *rc now has effective type uint32_t because a value is read
// from it with that type. The read operation is valid because we used calloc.
// Now the object pointed to by r (which is the same as *rc) has
// gained an effective type, although we didn't change its value.
uint32_t c = *rc;

// the object pointed to by s has no effective type, yet.
void* s = malloc(sizeof uint32_t);
// the object pointed to by s now has effective type uint32_t
// because an uint32_t value is copied into it.
memcpy(s, r, sizeof uint32_t);
```

Observe that for the latter, it was not necessary that we even have an `uint32_t*` pointer to that object. The fact that we have copied another `uint32_t` object is sufficient.

Section 36.2: restrict qualification

If we have two pointer arguments of the same type, the compiler can't make any assumption and will always have to assume that the change to `*e` may change `*f`:

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}

float fval = 4;
float eval = 77;
 fun(&eval, &fval);
```

all goes well and something like

> is 4 equal to 4?

is printed. If we pass the same pointer, the program will still do the right thing and print

> is 4 equal to 22?

This can turn out to be inefficient, if we *know* by some outside information that `e` and `f` will never point to the same data object. We can reflect that knowledge by adding `restrict` qualifiers to the pointer parameters:

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}
```

Then the compiler may always suppose that `e` and `f` point to different objects.

Section 36.3: Changing bytes

Once an object has an effective type, you should not attempt to modify it through a pointer of another type, unless that other type is a character type, `char`, `signed char` or `unsigned char`.

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

int main(void) {
  uint32_t a = 57;
  // conversion from incompatible types needs a cast !
  unsigned char* ap = (unsigned char*)&a;
  for (size_t i = 0; i < sizeof a; ++i) {
    /* set each byte of a to 42 */
    ap[i] = 42;
  }
  printf("a now has value %" PRIu32 "\n", a);
}
```

This is a valid program that prints

> a now has value 707406378

This works because:

- The access is made to the individual bytes seen with type `unsigned char` so each modification is well defined.
- The two views to the object, through `a` and through `*ap`, alias, but since `ap` is a pointer to a character type, the strict aliasing rule does not apply. Thus the compiler has to assume that the value of `a` may have been changed in the `for` loop. The modified value of `a` must be constructed from the bytes that have been changed.
- The type of `a`, `uint32_t` has no padding bits. All its bits of the representation count for the value, here

707406378, and there can be no trap representation.

Section 36.4: Character types cannot be accessed through non-character types

If an object is defined with static, thread, or automatic storage duration, and it has a character type, either: `char`, `unsigned char`, or `signed char`, it may not be accessed by a non-character type. In the below example a `char` array is reinterpreted as the type `int`, and the behavior is undefined on every dereference of the `int` pointer `b`.

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

    _Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

This is undefined because it violates the "effective type" rule, no data object that has an effective type may be accessed through another type that is not a character type. Since the other type here is `int`, this is not allowed.

Even if alignment and pointer sizes would be known to fit, this would not exempt from this rule, behavior would still be undefined.

This means in particular that there is no way in standard C to reserve a buffer object of character type that can be used through pointers with different types, as you would use a buffer that was received by `malloc` or similar function.

A correct way to achieve the same goal as in the above example would be to use a `union`.

```
typedef union bufType bufType;
union bufType {
   char c[sizeof(int[25])];
   int i[25];
};

int main( void )
{
    bufType a = { .c = { 0 } }; // reserve a buffer and initialize
    int* b = a.i;      // no cast necessary
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

    _Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

Here, the `union` ensures that the compiler knows from the start that the buffer could be accessed through different

views. This also has the advantage that now the buffer has a "view" `a.i` that already is of type `int` and no pointer conversion is needed.

Section 36.5: Violating the strict aliasing rules

In the following code let us assume for simplicity that `float` and `uint32_t` have the same size.

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g should equal %g\n", a, b);
}
```

`u` and `f` have different base type, and thus the compiler can assume that they point to different objects. There is no possibility that `*f` could have changed between the two initializations of `a` and `b`, and so the compiler may optimize the code to something equivalent to

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g should equal %g\n", a, a);
}
```

That is, the second load operation of `*f` can be optimized out completely.

If we call this function "normally"

```
float fval = 4;
uint32_t uval = 77;
fun(&uval, &fval);
```

all goes well and something like

> 4 should equal 4

is printed. But if we cheat and pass the same pointer, after converting it,

```
float fval = 4;
uint32_t* up = (uint32_t*)&fval;
fun(up, &fval);
```

we violate the strict aliasing rule. Then the behavior becomes undefined. The output could be as above, if the compiler had optimized the second access, or something completely different, and so your program ends up in a completely unreliable state.