Effective types: examples (extracts from P1796R0, plus more)

Peter Sewell, Kayvan Memarian, Victor B. F. Gomes,
Jens Gustedt, Hubert Tong

18 July 2019

WG21 UB: with notes from WG21 Cologne UB meeting

with thanks to other C memory object model people

Provenance and subobjects

Yesterday: only considered provenance at per-allocation granularity.

Provenance and subobjects: container-of casts

Can one cast from a pointer to the first member of a struct to the struct as a whole, and then use that to access other members.

Yes?
WG21 UB: Allowed

Provenance and subobjects: multidimensional arrays

ISO C: multidimensional arrays are recursively structured arrays-of-arrays.

For access via explicit indexing: yes.

For access via pointer arithmetic: should (e.g.) a linear traversal of a multidimensional array be allowed?

WG21 UB: Desired for medical imaging.
See p9 C++ casts between different array types are not allowed.
Jens: C has an example that it's not allowed
Hal: as a practical matter, it'd be very hard to optimise assuming you couldn't linearise; there's too much code.
Gaby: think multidimensional arrays collapse. "contiguous" is what allows pointer arithmetic.
Hubert: no, you have 1-past problems for all subarrays

Provenance and subobjects: representation-byte arithmetic and access

N2222 2.5.4 Q34 **Can one move among the members of a struct using representation-pointer arithmetic and casts?**

Yes

Can one move among the members of a struct with other pointer arithmetic?

```
// provenance_intra_object_1.c
#include <stdio.h>
#include <string.h>
typedef struct { int x; int y; } st;
int main() {
  st s = { .x=1, .y=2 };
  int *p = &s.x + 1;
  int *q = &s.y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // is this free of undefined behaviour?
    printf("s.x=%d s.y=%d *p=%d *q=%d\n",s.x,s.y,*p,*q);
  }
}
```

No

Plan?

Have a per-subobject provenance restriction by default, but relax this (to per-allocation provenance) for pointers that have been formed by an explicit cast.
Perhaps only for casts to `void *`, `unsigned char *`, `intptr_t`, or `uintptr_t`, or perhaps (for simplicity) for all casts.

Effective types

Q73. Can one do type punning between arbitrary types? No

Q91. Can a pointer to a structure alias with a pointer to one of its members? Yes

Q76. After writing a structure to a malloc'd region, can its members can be accessed via pointers of the individual member types? Yes

WG21 UB: Hubert: in C++ it's allowed but you need launder.

...lots of discussion about whether the C++ text actually allows this example

looking at the examples in `cmom-0004-2019-03-14-effective-types-examples.pdf`:

`effective_type_5.c` Hubert: uncontroversial

`effective_type_5d.c` does C++ need to magic up a pointer conversion in addition to magic'ing up an object?

Not allowed (without annotation) in current / future-planned text. Would need launder to give back a usable pointer of the type you want.

Q93. After writing all members of structure in a malloc'd region, can the structure be accessed as a whole? Yes

Q92. Can one do whole-struct type punning between distinct but isomorphic structure types in an allocated region?

A: Basic. This example writes a value of one struct type into a mallocâĂŹd region then reads it via a pointer to a distinct but isomorphic struct type.

```
// effective_type_2b.c
#include <stdio.h>
#include <stdlib.h>
typedef struct { int  i1; } st1;
typedef struct { int  i2; } st2;
int main() {
  void *p = malloc(sizeof(st1));
  st1 *p1 = (st1 *)p;
  *p1 = (st1){.i1 = 1};
  st2 *p2 = (st2 *)p;
  st2 s2 = *p2;      // undefined behaviour?
  printf("s2.i2=%i\n",s2.i2);
}
```

no

WG21 UB: ok. n C++ just doing the `p2->i2` is UB, even if you don't do the access

Q92. Can one do whole-struct type punning between distinct but isomorphic structure types in an allocated region?

B: read via lvalue merely at type **int**, but constructed via a pointer of type `st2 *`

```
   // effective_type_2d.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef struct { int  i1; } st1;
4  typedef struct { int  i2; } st2;
5  int main() {
6    void *p = malloc(sizeof(st1));
7    st1 *p1 = (st1 *)p;
8    *p1 = (st1){.i1 = 1};
9    st2 *p2 = (st2 *)p;
10   int *pi = &(p2->i2); // defined behaviour?
11   int i = *pi;         // defined behaviour?
12   printf("i=%i\n",i);
13 }
```

no?

WG21 UB: David: in C++ UB already on line 10

Q92. Can one do whole-struct type punning between distinct but isomorphic structure types in an allocated region?

C: read via an lvalue merely at type **int**, constructed by `offsetof` pointer arithmetic.

```
// effective_type_2e.c
#include <stdio.h>
#include <stdlib.h>
typedef struct { int  i1; } st1;
typedef struct { int  i2; } st2;
int main() {
  void *p = malloc(sizeof(st1));
  st1 *p1 = (st1 *)p;
  *p1 = (st1){.i1 = 1};
  st2 *p2 = (st2 *)p;
  int *pi = (int *)((char*)p + offsetof(st2,i1));
  int i = *pi;         // defined behaviour?
  printf("i=%i\n",i);
}
```

yes WG21 UB: This is the same as the `effective_type_5d`, which Hubert said needed launder (and Richard Smith agreed). Because `pf` is derived from `p`, not an `st1` there. So not allowed in C++ without launder. In C++ the check at lvalue construction time means that effective-type constraints don't have to remember lvalue construction.

Q92. Can one do whole-struct type punning between distinct but isomorphic structure types in an allocated region?

D: Here `f` is given aliased pointers to two distinct but isomorphic struct types, and uses them both to access an **`int`** member of a struct. We presume this is intended to be forbidden.
But the lvalue expressions, `s1p->i1` and `s2p->i2`, have identical type. this case. To forbid it, we have to take the construction of the lvalues into account, to see the types of `s1p` and `s2p`, not just the types of `s1p->i1` and `s2p->i2`.

```c
// effective_type_2.c
#include <stdio.h>
typedef struct { int  i1; } st1;
typedef struct { int  i2; } st2;
void f(st1* s1p, st2* s2p) {
  s1p->i1 = 2;
  s2p->i2 = 3;
  printf("f: s1p->i1 = %i\n",s1p->i1);
}
int main() {
  st1 s = {.i1 = 1};
  st1 * s1p = &s;
  st2 * s2p;
  s2p = (st2*)s1p;
  f(s1p, s2p); // defined behaviour?
```

Effective types and representation-byte writes

ISO C says: copying an object *"as an array of character type"* carries the effective type.
But should representation byte writes with other integers affect the effective type?

A: take the result of a `memcpy`'d **`int`** and then overwrite all of its bytes with zeros before trying to read it as an **`int`**.
allowed
WG21 UB: in Richard's paper, the object already existed (but not with user-memcpy)

B: similar, but tries to read the resulting memory as a **`float`** (presuming the implementation-defined fact that these have the same size and alignment, and that pointers to them can be meaningfully interconverted).
?
WG21 UB: Hubert: this would be allowed. The memcpy creates objects but not necessarily ones of the types you had.
Jens: C++ allows type punning through memcpy (but not via unions). C is the opposite.

Q75. Can an unsigned character array with static or automatic storage duration be used (in the same way as a 'malloc''d region) to hold values of other types?

ISO C: no. Real-world: yes ?

WG21 UB: in C++ it doesn't matter whether it was automatic/static or malloc'd

Hubert's examples

These show that current compiler behaviour is not consistent with the ISO C notion of effective types that allows type-changing updates within allocated regions simply by memory writes.

```
typedef struct A { int x, y; } A;
typedef struct B { int x, y; } B;

__attribute__((__noinline__, __weak__))
void f(long unk, void *pa, void *pa2, void *pb, long *x) {
  for (long i = 0; i < unk; ++i) {
    int oldy = ((A *)pa)->y;
    ((B *)pb)->y = 42;
    ((A *)pa2)->y = oldy ^ x[i];
  }
}
int main(void) {
  void *p = malloc(sizeof(A));
  ((A *)p)->y = 13;
  f(1, p, p, p, (long []){ 0 });
  printf("pa->y(%d)\n", ((A *)p)->y);
}
```

P0593R4 Implicit creation of objects for low-level object manipulation

The abstract machine creates objects of implicit lifetime types within those regions of storage as needed to give the program defined behavior.

(a) whole-program definedness is neither co- nor contra-variant: finding more executions may also find data races or unsequenced races

(b) style!

Accumulate constraints?