Effective types: examples (P1796R0)

PETER SEWELL, University of Cambridge
KAYVAN MEMARIAN, University of Cambridge
VICTOR B. F. GOMES, University of Cambridge
JENS GUSTEDT, INRIA
HUBERT TONG

This is a collection of examples exploring the semantics that should be allowed for objects and subobjects in allocated regions – especially, where the defined/undefined-behaviour boundary should be, and how that relates to compiler alias analysis. The examples are in C, but much should be similar in C++. We refer to the ISO C notion of effective types, but that turns out to be quite flawed. Some examples at the end (from Hubert Tong) show that existing compiler behaviour is not consistent with type-changing updates.

This is an updated version of part of n2294 C Memory Object Model Study Group: Progress Report, 2018-09-16.

1 INTRODUCTION

Paragraphs 6.5p{6,7} of the standard introduce *effective types*. These were added to C in C99 to permit compilers to do optimisations driven by type-based alias analysis, by ruling out programs involving unannotated aliasing of references to different types (regarding them as having undefined behaviour). However, this is one of the less clear, less well-understood, and more controversial aspects of the standard, as one can see from various GCC and Linux Kernel mailing list threads[1], blog postings[2], and the responses to Questions 10, 11, and 15 of our survey[3]. See also earlier committee discussion[4].

Moreover, the ISO text seems not to capture existing mainstream compiler behaviour. The ISO text (recalled below) is in terms of the types of the lvalues used for access, but compilers appear to do type-based alias analysis based on the construction of the lvalues, not just the types of the lvalues as a whole. Additionally, some compilers seem to differ from ISO in requiring syntactic visibility of union definitions in order to allow accesses to structures with common prefixes inside unions. The ISO text also leaves several questions unclear, e.g. relating to memory initialised piece-by-piece and then read as a struct or array, or vice versa.

Additionally, several major systems software projects, including the Linux Kernel, the FreeBSD Kernel, and PostgreSQL disable type-based alias analysis with the `-fno-strict-aliasing` compiler flag. The semantics of this (as for other dialects of C) is currently not specified by the ISO standard; it is debatable whether it would be useful to do that.

---

[1] https://gcc.gnu.org/ml/gcc/2010-01/msg00013.html, https://lkml.org/lkml/2003/2/26/158, and http://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg01647.html

[2] http://blog.regehr.org/archives/959, http://cellperformance.beyond3d.com/articles/2006/06/understanding-strict-aliasing.html, http://davmac.wordpress.com/2010/02/26/c99-revisited/, http://dbp-consulting.com/tutorials/StrictAliasing.html, and http://stackoverflow.com/questions/2958633/gcc-strict-aliasing-and-horror-stories

[3] https://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html (N2014), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2015.pdf (N2015)

[4] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1409.htm and http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1422.pdf (p14)

2019.

1.1 The ISO standard text

The C11 standard says, in 6.5:

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any[87]. If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using `memcpy` or `memmove`, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[88]

- a type compatible with the effective type of the object,
- a qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to the effective type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type.

Footnote 87) Allocated objects have no declared type.
Footnote 88) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

As Footnote 87 says, allocated objects (from `malloc`, `calloc`, and presumably any fresh space from `realloc`) have no declared type, whereas objects with static, thread, or automatic storage durations have some declared type.

For the latter, 6.5p{6,7} say that the effective types are fixed and that their values can only be accessed by an lvalue that is similar ("compatible", modulo signedness and qualifiers), an aggregate or union containing such a type, or (to access its representation) a character type.

For the former, the effective type is determined by the type of the last write, or, if that is done by a `memcpy`, `memmove`, or user-code char array copy, the effective type of the source.

2 EFFECTIVE TYPE EXAMPLES

2.1 Basic Effective Types

Q73. Can one do type punning between arbitrary types?

This basic example involves a write of a **`uint32_t`** that is read as a **`float`** (assuming that the two have the same size, and, unchecked in the code, that the latter does not require a stronger alignment constraint, and that casts between those two pointer types are implementation-defined to work). The example is clearly and uncontroversially forbidden by the standard text, and this fact is exploited by current compilers, which use the types of the arguments of `f` to reason that pointers `p1` and `p2` cannot alias.

```
// effective_type_1.c
```

```c
#include <stdio.h>
#include <inttypes.h>
#include <assert.h>
void f(uint32_t *p1, float *p2) {
  *p1 = 2;
  *p2 = 3.0; // does this have defined behaviour?
  printf("f: *p1 = %" PRIu32 "\n",*p1);
}
int main() {
  assert(sizeof(uint32_t)==sizeof(float));
  uint32_t i = 1;
  uint32_t *p1 = &i;
  float *p2;
  p2 = (float *)p1;
  f(p1, p2);
  printf("i=%" PRIu32 "  *p1=%" PRIu32
         "  *p2=%f\n",i,*p1,*p2);
}
```

With `-fstrict-aliasing` (the default for GCC), GCC assumes in the body of `f` that the write to `*p2` cannot affect the value of `*p1`, printing `2` (instead of the integer value of the representation of `3.0` that would the most recent write in a concrete semantics): while with `-fno-strict-aliasing` it does not assume that. The former behaviour can be justified by regarding the program as having undefined behaviour, due to the write of the **`uint32_t`** `i` with a **`float`** lvalue.

2.2 Structs and their members

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

In this example `f` is given a pointer to a struct and an aliased pointer to its first member, writing via the struct pointer and reading via the member pointer. We presume this is intended to be allowed. The ISO text permits it if one reads the first bullet *"a type compatible with the effective type of the object"* as referring to the **`int`** subobject of `s` and not the whole `st` typed object `s`, but the text is generally unclear about the status of subobjects.

```c
// effective_type_2c.c
#include <stdio.h>
typedef struct { int i; } st;
void f(st* sp, int* p) {
  sp->i = 2;
  *p = 3;
  printf("f: sp->i=%i  *p=%i\n",sp->i,*p); // prints 3,3 not 2,3 ?
}
int main() {
  st s = {.i = 1};
  st *sp = &s;
  int *p = &(s.i);
  f(sp, p);
  printf("s.i=%i  sp->i=%i  *p=%i\n", s.i, sp->i, *p);
}
```

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

The examples below write a struct into a malloc'd region then read one of its members, first using a a pointer constructed using `char *` arithmetic, and then cast to a pointer to the member type, and second constructed from `p` cast to a pointer to the struct type.

We presume both should be allowed.

The types of the lvalues used for the member reads are the same, so by the 6.5p6,7 text this should make no difference, but a definition of effective types that matches current TBAA practice, by taking lvalue construction into account, may need to take care to permit this.

```
// effective_type_5.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
int main() {
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  st1 s1 = { .c1='A', .f1=1.0};
  *((st1 *)p) = s1;
  float *pf = &(((st1 *)p)->f1);
  // is this free of undefined behaviour?
  float f = *pf;
  printf("f=%f\n",f);
}
```

```
// effective_type_5d.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
int main() {
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  st1 s1 = { .c1='A', .f1=1.0};
  *((st1 *)p) = s1;
  float *pf = (float *)((char*)p + offsetof(st1,f1));
  // is this free of undefined behaviour?
  float f = *pf;
  printf("f=%f\n",f);
}
```

Q93. After writing all members of structure in a malloc'd region, can the structure be accessed as a whole? Our reading of C11 and proposal for C2x: C11: yes (?)

The examples below write the members of a struct into a malloc'd region and then read the struct as a whole. In the first example, the lvalues used for the member writes are constructed using `char *` arithmetic, and then cast to the member types, while in the second, they are constructed from `p` cast to a pointer to the struct type.

Similarly to Q76 above, the types of the lvalues used for the member writes are the same, so by the 6.5p6,7 text this should make no difference, but a definition of effective types that matches current TBAA practice, by taking lvalue construction into account, may need to take care to permit this.

```
// effective_type_5c.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
int main() {
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  char *pc = &((*((st1*)p)).c1);
  *pc = 'A';
  float *pf = &((*((st1*)p)).f1);
  *pf = 1.0;
  st1 *pst1 = (st1 *)p;
  st1 s1;
  s1 = *pst1;   // is this free of undefined behaviour?
  printf("s1.c1=%c  s1.f1=%f\n", s1.c1, s1.f1);
}
```

```
// effective_type_5b.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
int main() {
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  char *pc = (char*)((char*)p + offsetof(st1, c1));
  *pc = 'A';
  float *pf = (float *)((char*)p + offsetof(st1,f1));
  *pf = 1.0;
  st1 *pst1 = (st1 *)p;
  st1 s1;
  s1 = *pst1;   // is this free of undefined behaviour?
  printf("s1.c1=%c  s1.f1=%f\n", s1.c1, s1.f1);
}
```

2.3 Isomorphic Struct Types

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

This example writes a value of one struct type into a malloc'd region then reads it via a pointer to a distinct but isomorphic struct type.

We presume this is intended to be forbidden. The ISO text is not clear here, depending on how one understands subobjects, which are not well-specified.

```
// 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);
}
```

The above test discriminates between a notion of effective type that only applies to the leaves, and one which takes struct/union types into account.

The following variation does a read via an lvalue merely at type **int**, albeit with that lvalue constructed via a pointer of type `st2 *`. This is more debatable. For consistency with the apparent normal implementation practice to take lvalue construction into account, it should be forbidden.

```
// effective_type_2d.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 = &(p2->i2); // defined behaviour?
  int i = *pi;         // defined behaviour?
  printf("i=%i\n",i);
}
```

The following variation does a read via an lvalue merely at type **int**, constructed by `offsetof` pointer arithmetic. This should presumably be allowed.

```
// 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);
}
```

Q74. Can one do type punning between distinct but isomorphic structure types?

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, and GCC appears to assume that it is, printing `f: s1p->i1 = 2`.

However, the two lvalue expressions, `s1p->i1` and `s2p->i2`, are both of the identical (and hence "compatible") **int** type, so the ISO text appears to allow this case. To forbid it, we have to somehow 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`.

```
// 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?
  printf("s.i1=%i  s1p->i1=%i  s2p->i2=%i\n",
         s.i1,s1p->i1,s2p->i2);
}
```

2.4 Isomorphic Struct Types – additional examples

It's not clear whether these add much to the examples above; if not, they should probably be removed.

Q80. After writing a structure to a malloc'd region, can its members be accessed via a pointer to a different structure type that has the same leaf member type at the same offset?

```
// effective_type_9.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
typedef struct { char c2; float f2; } st2;
int main() {
  assert(sizeof(st1)==sizeof(st2));
  assert(offsetof(st1,c1)==offsetof(st2,c2));
  assert(offsetof(st1,f1)==offsetof(st2,f2));
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  st1 s1 = { .c1='A', .f1=1.0};
  *((st1 *)p) = s1;
  // is this free of undefined behaviour?
  float f = ((st2 *)p)->f2;
  printf("f=%f\n",f);
}
```

Q94. After writing all the members of a structure to a malloc'd region, via member-type pointers, can its members be accessed via a pointer to a different structure type that has the same leaf member types at the same offsets?

```
// effective_type_9b.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
typedef struct { char c2; float f2; } st2;
int main() {
```

```
  assert(sizeof(st1)==sizeof(st2));
  assert(offsetof(st1,c1)==offsetof(st2,c2));
  assert(offsetof(st1,f1)==offsetof(st2,f2));
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  char *pc = (char*)((char*)p + offsetof(st1, c1));
  *pc = 'A';
  float *pf = (float *)((char*)p + offsetof(st1,f1));
  *pf = 1.0;
  // is this free of undefined behaviour?
  float f = ((st2 *)p)->f2;
  printf("f=%f\n",f);
}
```

Here there is nothing specific to `st1` or `st2` about the initialisation writes, so the read of `f` should be allowed.

```
// effective_type_9c.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
typedef struct { char c2; float f2; } st2;
int main() {
  assert(sizeof(st1)==sizeof(st2));
  assert(offsetof(st1,c1)==offsetof(st2,c2));
  assert(offsetof(st1,f1)==offsetof(st2,f2));
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  st1 *pst1 = (st1*)p;
  pst1->c1 = 'A';
  pst1->f1 = 1.0;
  float f = ((st2 *)p)->f2; // is this free of undefined behaviour?
  printf("f=%f\n",f);
}
```

Here the construction of the lvalues used to write the structure members involves `st1`, but the lvalue types do not. The 6.5p6,7 text is all in terms of the lvalue types, not their construction, so in our reading of C11 this is similarly allowed.

2.5 Effective types and representation-byte writes

The ISO text explicitly states that copying an object *"as an array of character type"* carries the effective type across:

"If a value is copied into an object having no declared type using `memcpy` *or* `memmove`*, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one."*

The first two examples below should therefore both be allowed, using `memcpy` to copy from an **`int`** in a local variable and in a malloc'd region (respectively) to a malloc'd region, and then reading that with an **`int*`** pointer.

```
// effective_type_4b.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
```

```
    int i=1;
    void *p = malloc(sizeof(int));
    memcpy((void*)p, (const void*)(&i), sizeof(int));
    int *q = (int*)p;
    int j=*q;
    printf("j=%d\n",j);
  }
```

```
  // effective_type_4c.c
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  int main() {
    void *o = malloc(sizeof(int));
    *(int*)o = 1;
    void *p = malloc(sizeof(int));
    memcpy((void*)p, (const void*)o, sizeof(int));
    int *q = (int*)p;
    int j=*q;
    printf("j=%d\n",j);
  }
```

The following variant of the first example should also be allowed, copying as an unsigned character array rather than with the library `memcpy`.

```
  // effective_type_4d.c
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  void user_memcpy(unsigned char* dest,
                   unsigned char *src, size_t n) {
    while (n > 0)  {
      *dest = *src;
      src += 1; dest += 1; n -= 1;
    }
  }
  int main() {
    int i=1;
    void *p = malloc(sizeof(int));
    user_memcpy((unsigned char*)p, (unsigned char*)(&i), sizeof(int));
    int *q = (int*)p;
    int j=*q;
    printf("j=%d\n",j);
  }
```

Should representation byte writes with other integers affect the effective type? The first example below takes the result of a `memcpy`'d **`int`** and then overwrites all of its bytes with zeros before trying to read it as an **`int`**. The second is similar, except that it 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). The first should presumably be allowed. It is unclear to us whether the second should be allowed or not.

```
  // effective_type_4e.c
  #include <stdio.h>
  #include <stdlib.h>
```

```
#include <string.h>
int main() {
  int i=1;
  void *p = malloc(sizeof(int));
  memcpy((void*)p, (const void*)(&i), sizeof(int));
  int k;
  for (k=0;k<sizeof(int);k++)
    *(((unsigned char*)p)+k)=0;
  int *q = (int*)p;
  int j=*q;
  printf("j=%d\n",j);
}
```

```
// effective_type_4f.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
int main() {
  int i=1;
  void *p = malloc(sizeof(int));
  memcpy((void*)p, (const void*)(&i), sizeof(int));
  int k;
  for (k=0;k<sizeof(int);k++)
    *(((unsigned char*)p)+k)=0;
  int *q = (int*)p;
  assert(sizeof(float)==sizeof(int));
  assert(_Alignof(float)==_Alignof(int));
  float f=*q;
  printf("f=%f\n",f);
}
```

2.6 Unsigned character arrays

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?

This seems to be forbidden by the ISO text, but we believe it is common in practice. Question 11 of our survey relates to this.

A literal reading of the effective type rules prevents the use of an unsigned character array as a buffer to hold values of other types (as if it were an allocated region of storage). For example, the following has undefined behaviour due to a violation of 6.5p7 at the access to `*fp`. (This reasoning relies on the implementation-defined property that the conversion of the `(float * )c` cast gives a usable result – the conversion is permitted by 6.3.2.3p7 but the standard text only guarantees a roundtrip property.)

```
// effective_type_3.c
#include <stdio.h>
#include <stdalign.h>
int main() {
  _Alignas(float) unsigned char c[sizeof(float)];
  float *fp = (float *)c;
  *fp=1.0; // does this have defined behaviour?
  printf("*fp=%f\n",*fp);
```

```
}
```

Even bytewise copying of a value via such a buffer leads to unusable results in the standard:

```
// effective_type_4.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdalign.h>
int main() {
  _Alignas(float) unsigned char c[sizeof(float)];
  // c has effective type char array
  float f=1.0;
  memcpy((void*)c, (const void*)(&f), sizeof(float));
  // c still has effective type char array
  float *fp = (float *) malloc(sizeof(float));
  // the malloc'd region initially has no effective type
  memcpy((void*)fp, (const void*)c, sizeof(float));
  // does the following have defined behaviour?
  // (the ISO text says the malloc'd region has effective
  // type unsigned char array, not float, and hence that
  // the following read has undefined behaviour)
  float g = *fp;
  printf("g=%f\n",g);
}
```

This seems to be unsupportable for a systems programming language: a character array and malloc'd region should be interchangeably usable, either on-demand or by default. GCC developers commented that they essentially ignore declared types in alias analysis because of this.

For C2X, we believe there has to be some (local or global) mechanism to allow this.

2.7 Overlapping structs in malloc'd regions

Q79. After writing one member of a structure to a malloc'd region, can a member of another structure, with footprint overlapping that of the first structure, be written?

```
// effective_type_8a.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
typedef struct { char c2; float f2; } st2;
int main() {
  assert(sizeof(st1)==sizeof(st2));
  assert(offsetof(st1,c1)==offsetof(st2,c2));
  assert(offsetof(st1,f1)==offsetof(st2,f2));
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  ((st1 *)p)->c1 = 'A';
  // is this free of undefined behaviour?
  ((st2 *)p)->f2 = 1.0;
  // is this defined, and always prints 'A' ?
  printf("((st1 *)p)->c1 = '%c'\n", ((st1 *)p)->c1);
```

```
}
```

```
// effective_type_8b.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
typedef struct { char c2; float f2; } st2;
int main() {
  assert(sizeof(st1)==sizeof(st2));
  assert(offsetof(st1,c1)==offsetof(st2,c2));
  assert(offsetof(st1,f1)==offsetof(st2,f2));
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  ((st1 *)p)->c1 = 'A';
  // is this free of undefined behaviour?
  ((st2 *)p)->f2 = 1.0;
  // is this defined, and always prints 'A' ?
  printf("((st2 *)p)->c2 = '%c'\n", ((st2 *)p)->c2);
}
```

Again this is exploring the effective type of the footprint of the structure type used to form the lvalue. We presume this should be allowed – from one point of view, it is just a specific instance of the strong (type changing) updates that C permits in malloc'd regions.

2.8 Effective types and uninitialised reads

Q77. Can a non-character value be read from an uninitialised malloc'd region?

```
// effective_type_6.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
int main() {
  void *p = malloc(sizeof(float)); assert (p != NULL);
  // is this free of undefined behaviour?
  float f = *((float *)p);
  printf("f=%f\n",f);
}
```

The effective type rules seem to deem this undefined behaviour.

```
// effective_type_6b.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
int main() {
  void *p = calloc(1, sizeof(float)); assert (p != NULL);
  // is this free of undefined behaviour?
  float f = *((float *)p);
  printf("f=%f\n",f);
}
```

For this variant where `calloc` does initialise to zero, Jens suggests the program should be well defined (but the current standard text still makes this undefined).

Q78. After writing one member of a structure to a malloc'd region, can its other members be read?

```
// effective_type_7.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; unsigned int ui1; } st1;
int main() {
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  ((st1 *)p)->c1 = 'A';
  // is this free of undefined behaviour?
  unsigned int ui = ((st1 *)p)->ui1;
  printf("ui=%d\n",ui);
}
```

If the write should be considered as affecting the effective type of the footprint of the entire structure, then it would change the answer to `effective_type_5.c` here. It seems unlikely but not impossible that such an interpretation is desirable.

There is a defect report (which?) about copying part of a structure and effective types.

2.9 Properly overlapping objects

Q81. Can one access two objects, within a malloc'd region, that have overlapping but non-identical footprint?

Robbert Krebbers asks on the GCC list (https://gcc.gnu.org/ml/gcc/2015-03/msg00083.html) whether "GCC uses 6.5.16.1p3 of the C11 standard as a license to perform certain optimizations. If so, could anyone provide me an example program. In particular, I am interested about the 'then the overlap shall be exact' part of 6.5.16.1p3: *If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.*" Richard Biener replies with this example (rewritten here to print the result), saying that it will be optimised to print 1 and that this is basically effective-type reasoning.

```
// krebbers_biener_1.c
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
struct X { int i; int j; };
int foo (struct X *p, struct X *q) {
  // does this have defined behaviour?
  q->j = 1;
  p->i = 0;
  return q->j;
}
int main() {
  assert(sizeof(struct X) == 2 * sizeof(int));
  unsigned char *p = malloc(3 * sizeof(int));
  printf("%i\n", foo ((struct X*)(p + sizeof(int)),
```

```
                        (struct X*)p));
}
```

2.10 Examples from Jens' visit

The interaction between out-of-bound pointer arithmetic checks (at the level of subobject) and unions is problematic. In the following, a choice needs to be made regarding which subobject is being accessed by the last line of the main function. If it is the array inside the first member of the union, the access is out of bound. But if it is the array in the second member of union, this program is well defined.

```
// effective_type_jens_1.c
struct T{
  union U {
    struct T1 {
      int x[2];
    } st1;
    struct T2 {
      int y[3];
    } st2;
  } un;
  char c;
} z;

int main(void)
{
  int *p = (int*)( (char*)(&(z.c)) - offfsetof(struct T, c) );
  p[2] = 10; // this is a defined access to z.un.st2.y[2] ?
}
```

One could think of making the semantics "angelic", but the following variant shows it is not clear how to do so.

```
// effective_type_jens_1b.c
struct T{
  union U {
    struct T1 {
      int x[2];
      int y[3];
    } st1;
    struct T2 {
      int x[3];
      int y[2];
    } st2;
  } un;
  char c;
} z;

int main(void)
{
  int *p = (int*)( (char*)(&(z.c)) - offfsetof(struct T, c) );
  p[2] = 10; // what's happening here?
}

// effective_type_jens_2.c
struct S {
```

```
  int x;
};

struct T {
  int y;
};

int main(void)
{
  int *p = malloc(sizeof *p);
  *p = 5;

  struct T *q = (struct T*)p;
  q->y = 10;

  struct S *s = (struct S*)p; // is s a valid pointer to a fully initialised "struct S" object?
}
```

2.11 Hubert's examples

These examples 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.

This was his first example:

```
// effective_type_hubert_1.c
#include <stdlib.h>
#include <string.h>

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

//__attribute__((__noinline__, __weak__))
B *newB(void *p) {
    static const B b = { 0 };
    return (B *)memcpy(p, &b, sizeof b);
}

int main(void) {
  static const A a = { 0 };

  A *const ap = (A *)malloc(sizeof a);
  memcpy(ap, &a, sizeof a);

  B *const bp = newB(ap);
  bp->y = 42;
  ap->y = 0; // Hubert says: I think this should be UB.
             // Both Clang and GCC will not expect
             // this to alias bp->y under TBAA.
  return bp->y; // 42?
}
```

This was his example from the 2019-05-28 teleconf:

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

```
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);
}
```

He tried gcc and clang on x86 and POWER, with optimisation. The compiler thinks the write to `{(B*)pb)->y` is constant, so compilers lift or sink out of loop, clobbering the value of `((A *)p)->y`, because they assume `(B*)pb` and `(A*)pa` / `(A*)pa2` don't alias.

Changing to an int/float case makes no difference, so one couldn't work around this by just looking at the access types:

How could we proceed? Some conceivable options are below, but many look quite unappealing.

(1) add syntactic clues for type-changing updates, along the lines of the C++ placement new and launder
(2) provide an analogue of `-fno-strict-aliasing` on finer granularities
(3) specify the two dialects, with and without `-fno-strict-aliasing`, with the latter having a blanket prohibition on type-changing updates
(4) change compilers to remove this use of TBAA
(5) make function arguments all restrict
(6) use Hal's TBAA sanitiser (or other tools) to survey how common type-changing updates are in practice, in code bases that are compiled without `-fno-strict-aliasing`
(7) add new qualifier that lets one say the compiler can assume the types don't change