N2929: Memory layout of union members, v. 2.

February 6, 2022

Author: Javier A. Múgica
Reference: N2788. https://www9.open-std.org/JTC1/SC22/WG14/www/docs/n2788.pdf

The paper N2788 argues that the current standard does not force the layout of unions to be as intended. When this paper was discussed it seems that my comments regarding union pointers were not completely understood. A detailed example is provided in the second part of this document of a union's layout and the behaviour of related pointers.

More important is the fact that when I proposed a solution in that paper I was not aware of the special provision introduced by the standard for accessing objects of any type via pointer to character types (6.3.2.3, p. 7):

[...] When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object.

This simplifies the wording needed for forcing the layout of unions to be as intended. I no longer propose a rewording of a paragraph in 6.2.5 but the addition of a sentence to 6.7.2.1 p. 18, thereby paralleling the description in p. 17 for structures.

The specifications for unions go side by side with that for structures. Section 6.7.2.1, p. 7 reads:

As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

This is merely a remainder of what has been said in 6.2.5, stating in an unprecise form, as in 6.2.5, that a structure's members are allocated in an ordered sequence and those of a union overlap. The following is the referred to text from 6.2.5:

— A structure type describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

— A union type describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

For structures there follows, in 6.7.2.1, a precise specification in paragraph 17:

Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

But there is not such a precise description in the corresponding paragraph 18 for unions. Of paragraph 17 three clauses paragraph 18 only includes the one corresponding to the intermediate one. In particular, p. 18 is lacking

— The way union members overlap.
— That there may not be padding at its beginning.

It was pointed by Jens Gunsted that my proposed wording in N2788 did not preclude the existence of leading padding bytes. And he was right. He and others proposed "to change paragraph 17 in 6.7.2.1 to clarify that "structure object" includes or applies to unions, explicitly". It should be clear now that that is not the right solution, but to add the corresponding sentence to paragraph 18, for unions.

My final proposal is therefore to add two sentences to that p 18:

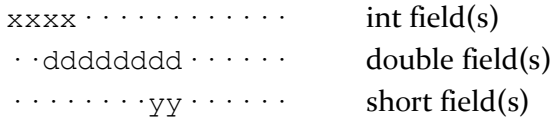
The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa. The members of a union object overlap in such a way that pointers to them when converted to pointers to character types point to the same byte. There may be unnamed padding at the end of a union object, but not at its beginning.

Straw poll: Should paragraph 18 in 6.7.2.1 be modified like this?

```
* --- * --- * --- * --- * --- * --- * --- * --- * --- * --- * --- *
```

An example union object

Here an example is presented to show that the addition of those sentences is necessary.



Let the union be named "u" and have members named u.i, u.d and u.s respectively. Suppose further that the int field(s) starts at address 1000. The union has six padding bytes at the end.

For pointers of type int (and double, and short), the values 1000-1015 all represent the same address, since if *k* is any of those values (int*)*k* will always point to the same integer (and (double*) to the same double and (short*) to the same short). The translator may choose arbitrarily and even non-deterministically any of them for representing a pointer of type int* (and double* and short*). That random value is converted always to 1000 (or 1002 or 1008) if the pointer is converted to *char-type** or void*, but need not for uintptr_t, since only conversion back to the same pointer type is meaningful in that case. The same can be said for a pointer to the union object itself.

This layout would be impossible for unions containing members of character type, but is still possible for a union like the one taken as example. Also, the possibility of using different numbers to represent a pointer to the same object of a certain type in the same way as is done in this example already exists: a pointer to an object of a type which requires an alignment of four can be represented by any of the four addresses from its actual address to the next possible one. In my example, the short within the union still occupies two bytes in the abstract machine (otherwise the example would be wrong) since, for instance, memcpy(u.s, b, 2) does copy all the short. It just has more possibilities for the number representing its address than an ordinary short.

```
int *p = 1003;
short *q = (short*)p;   //q stores some number in the range 1000 – 1015.
uintptr_t a = 1000, b = 1005;
if((double*)a == (double*)b);   //Compare equal
```

That the actual memory storing of the objects need not be as specified for the abstract machine is well known and poses no problem as long as it is invisible to the program. For example, the bytes of an array `int r[20]` could be stored in three separate pieces:

```
#####       ###     ############
```

Since the standard specifies that “An array type describes a contiguously allocated nonempty set of objects with a particular member object type, called the element type”, the program must behave in all ways as though the ints were stored contiguously. So for instance

```
memcpy(r,s,20*sizeof(int))
```

must copy all the array.

This is not the case for unions, since their layout is fully specified. Thus, continuing with the example, the instructions

```
int *q = &u.i;
memcpy(q,buffer,sizeof(double));
```

do not copy the whole union as one would expect (except for the trailing padding bytes), but just its first 8 bytes.

More example code, where the translator always chooses the value 1000:

```
short *q = &u.s;     // q = 1000.
*(double *)q = 1.0; // Correctly sets the double field(s)
*(int *)q = 2;       // Correctly sets the int field(s)
double *p = &u.d;    // p = 1000
short *q = &u.s;     // q = 1000
char *cd = (char *)p;           // cd = 1002, because p is if type double*.
char *cs = (char *)q;           // cs = 1008, because q is if type short*.
char *ci = (char *)(int *)p;    // ci = 1000, because (int*)p is if type int*
ci[2] = 37;          //Modifies the third byte of u.i.
```

The standard also requires that the char* pointer be convertible back to the original pointer and the result be the same. This can obviously be satisfied in this example. Note that there is no requirement that the pointer (ci+2), say, be convertible to anything related to the original union.

Let's review that this union satisfies all the requirements of the current standard.

1. A pointer to a union object, suitably converted, points to each of its members. YES

2. A pointer to the union or to each of its members may be converted to a void* pointer or to a uinptr_t integer and back and the result compares equal to the original pointer. YES

3. Each of the fields is represented in the same way as any other object of its same type: "Values stored in non-bit-field objects of any other object type consist of $n \times$ CHAR_BIT bits, where n is the size of an object of that type, in bytes. The value may be copied into an object of type unsigned char [n] (e.g., by memcpy)." YES

   For example,

   ```
   double b;
   memcpy(&u.d, b, sizeof(double));
   ```

   copies the value of the double field into b.

4. When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object. YES

5. The different fields honour the alignments for their respective types. YES

   Alignment manifests itself by the interconvertibility of pointers: "A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined. Otherwise, when

converted back again, the result shall compare equal to the original pointer." For example, it may happen that given an array `short x[20]`, of the series of pointers `x, x+1, x+2, x+3 ...` only one out of four is convertible to double*.

In the example union all pointers to its members and the union itself may be converted to the latter, double*, int* and (at least) any other type with an alignment requirement less than or equal to the maximum of these, and back, and the result will compare equal to the original pointer.

The example union, with its padding bytes at the end, is supposed to have an alignment of 16 bytes. The value of 1000 for the address is just a tag. The divisibility by powers of two of the integer numbers used to represent addresses need not be the alignment itself, though in practice it usually is.

6. A union type describes an overlapping nonempty set of member objects. YES

7. All pointers to members of the same union object compare equal. YES

8. If *p* is a pointer to any of its members the pointer *p* + 1 is valid and is indeed *p* + 1. YES

   For example,

```
int *p = (int*)&u.i;
(p+1) - p;      // Equals 1.
```

   The translator can insert a runtime check so that, if the pointers being substracted point the second to somewhere within a union object and the first to just past the union, and they are not pointers to a character type, the result of the substraction is one.

Note that a translator may insert enough information to know at run-time the whole meaning of all the program's bytes, as debuggers (may) do. It may even make the program keep track of all it execution history, if it so desired.

Even after my proposed addition to p. 18 the translator could insist that pointers to u.i and u.d, say, compare equal after convertion to char*, for it could make the program remember at run-time that the latter pointers derive from the former ones. But then there would be two pointers of type char* that compare equal and that point to different bytes, as could be made manifest in the abstract machine by modifying the byte pointed to by one of them and noting that the stored value does not show up in the other byte, and this is not permitted.

My original proposal was to replace paragraph 7 in "6.2.6 Representation of types" by

If two members A and B of an object of union type have sizes *m* and *n* respectively, with $m \leq n$, the first *m* bytes of the object representation of B share the same memory space as the *m* bytes of the object representation of A, in the same order, and are said to *overlap*. For this reason, when

a value is stored in a member of an object of union type, the bytes of the object representation of any other member that overlap with bytes of the object representation of the stored member take the same values. The bytes that do not overlap take unspecified values.

As has been explained, in view of the second half of 6.3.2.3, p. 7 now I advocate for a different solution and do not propose this paragraph for inclusion, not even as a footnote.