

# Re: Unions Redux

---

*Source:* [http://coding.derkeiler.com/Archive/C\\_CPP/comp.lang.c/2007-03/msg02601.html](http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2007-03/msg02601.html)

---

- *From:* Jack Klein <[jackklein@xxxxxxxxxxxx](mailto:jackklein@xxxxxxxxxxxx)>
  - *Date:* Wed, 14 Mar 2007 22:05:25 -0500
- 

On 14 Mar 2007 15:10:44 -0700, "Old Wolf" <[oldwolf@xxxxxxxxxxxx](mailto:oldwolf@xxxxxxxxxxxx)> wrote in comp.lang.c:

Ok, we've had two long and haphazard threads about unions recently, and I still don't feel any closer to certainty about what is permitted and what isn't. The other thread topics were "Real Life Unions" and "union { unsigned char u[10]; ... }".

Most of the rambling was caused by the original OP, I think, rather than the material. I am not criticizing, just observing.

Here's a concrete example:

```
#include <stdio.h>

int main(void)
{
    union { int s; unsigned int us; } u;

    u.us = 50;
    printf("%d\n", u.s);
    return 0;
}
```

Is this program well-defined (printing 50), implementation-defined, or UB ?

The program is well-defined, I'll elaborate further down.

Note that the aliasing rules in C99 6.5 are not violated here — it is not forbidden under that section to access an object of some type T with an lvalue expression whose type is the signed or unsigned version of T.

## Re: Unions Redux

As you pointed out, it does not violate the alias rules, but there are other rules to consider. In this particular case, you are accessing an object of type unsigned int with an lvalue expression of type signed int. The entire question of the validity of the operation depends on the object representation compatibility, and has nothing at all to do with the fact that there is a union involved.

In this particular case, the operation is well-defined because of the standard's guarantees about corresponding signed and unsigned integer types. For a positive value within the range of both types, the bit representation is identical for both.

However, if you had assigned `INT_MAX + 1` to `u.us`, and your implementation is one of the universal ones where `UINT_MAX > INT_MAX`, the behavior would be implementation-defined because, at least theoretically, `(unsigned)INT_MAX + 1` could contain a bit pattern that is a trap representation for signed int.

In other words, is there anything other than the aliasing rules that restrict 'free' use of unions?

Personally, I think people get to wound up in the mystical and magical properties of unions.

They are good for two things:

1. Space saving, such as a struct containing a data type specifier and a union of all the possible data types. This is a frequent feature in message passing systems. Generally, type punning is not used here.
2. Another way to do type punning.

Consider:

```
int test_lone(long l)
{
    int *ip = (int *)&l;
    int i = *ip;
    return i==l;
}
```

Is this code undefined, implementation-defined, or unspecified?

Technically it is undefined, but on an implementation like today's typical desktop, where `int` and `long` have the same representation and alignment, the result will be that the function returns 1. On an implementation where `int` and `long` are different sizes, who knows.

Now consider:

```
int test_long(long l)
{
union { long ll; int ii } li;
li.ll = l;
return li.ll==li.ii;
}
```

Is this code undefined? Well, yes, but it is no different in functionality than the first function. If int and long have the same representation, it will return 1.

There is no difference in aliasing in a union than there is via pointer casting.

--

Jack Klein

Home: <http://JK-Technology.Com>

FAQs for

comp.lang.c <http://c-faq.com/>

comp.lang.c++ <http://www.parashift.com/c++-faq-lite/>

alt.comp.lang.learn.c-c++

<http://www.contrib.andrew.cmu.edu/~ajo/docs/FAQ-acllc.html>

.