Unrestricted Unions

Authors: **Alan Talbot**
alan.talbot@teleatlas.com

Lois Goldthwaite
Lois@LoisGoldthwaite.com

Lawrence Crowl
Lawrence@Crowl.org
crowl@google.com

Document number: **N2412=07-0272**
Date: 2007-09-10

Project: Programming Language C++
Reference: N2369=07-0229

Abstract

This paper proposes a very small change to the language: the removal of some of the restrictions on members of unions. This change will make unions easier to use, more powerful, and much more useful. This change does not break any existing code since it only involves relaxing certain rules.

Proposed Wording

7.1.7 Alignment specifier [dcl.align]

10 [Note: the aligned_union template (20.4.7) can be used to create a union containing a type with a non-trivial ~~constructor or~~ destructor. —end note]

8.5 Initializers [dcl.init]

5 To zero-initialize an object of type T means:

— if T is a scalar type (3.9), the object is set to the value 0 (zero), taken as an integral constant expression, converted to T;[91)]

— if T is a non-union class type, each non-static data member and each base-class subobject is zero-initialized;

— if T is a union type, the object's first named data member[92)] is zero-initialized;

— if T is an array type, each element is zero-initialized;

— if T is a reference type, no initialization is performed.

To default-initialize an object of type T means:

— if T is a non-trivial class type (clause 9), the default constructor for T is called (and the initialization is ill-formed if T has no accessible default constructor);

— if T is an array type, each element is default-initialized;

— otherwise, the object is zero-initialized.

To value-initialize an object of type T means:

— if T is a class type (clause 9) with a user-provided constructor (12.1), then the default constructor for T is called (and the initialization is ill-formed if T has no accessible default constructor);

— if T is a non-union class type without a user-provided constructor, then every non-static data member and base-class component of T is value-initialized;[93)]

— if T is a union type without a user-provided constructor, the object's first named data member[92)] is value-initialized;[93)]

— if T is an array type, then each element is value-initialized;

— otherwise, the object is zero-initialized

9.5 Unions [class.union]

1 In a union, at most one of the data members can be active at any time, that is, the value of at most one of the data members can be stored in a union at any time. [Note: one special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (9.2), and if an object of this standard-layout union type contains one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of standard-layout struct members; see 9.2. —end note] The size of a union is sufficient to contain the largest of its data members. Each data member is allocated as if it were the sole member of a struct. A union can have member functions (including constructors and destructors), but not virtual (10.3) functions. A union shall not have base classes. A union shall not be used as a base class. An object of a non-trivial class may be a member of a union only if it has a trivial destructor (clause 9). An ~~object~~ array of ~~a~~ non-trivial class objects (clause 9) shall not be a member of a union~~, nor shall an array of such objects~~. If a union contains a static data member, or a member of reference type, the program is ill-formed. [Note: An implicitly defined default constructor (12.1), copy constructor (12.8) or assignment operator (12.8) will default construct, copy construct, or copy assign the first named data member. —end note]

[Example: Consider a union u of type U containing a member m of type M. If M is not trivially constructible (for instance, if it declares or inherits virtual functions), the member m can still be made active using the placement new operator and the non-trivial default constructor (or copy constructor) as follows:

```
new (&u.m) M;
```

The old member need not be explicitly destructed because by definition it has a trivial destructor. --end example]

12.1 Constructors [class.ctor]

~~11 A union member shall not be of a class type (or array thereof) that has a non-trivial constructor.~~

Motivation

When confronted with functionality that can be misused, a language design must often make a choice between prohibiting it in the interests of safety, and allowing it in the interests of flexibility and power. Generally speaking C++ takes the latter approach. However, in the case of unions C++ diverges from this philosophy and places severe limitations on members in the interest of safety. This is both inconsistent and unfortunate, because unions are very useful devices in some situations.

Many important problem domains require either large numbers of objects or very limited memory resources. In these situations conserving space is very important, and a union is often a perfect way to do that. In fact a common use case is the situation where a union never changes its active member during its lifetime. It can be constructed, copied, and destructed as if it were a struct containing only one member.

Unfortunately most objects cannot be members of unions. Good object programming practice results in constructors for even simple, lightweight, transparent types. For example a point type for geographic work needs a good set of constructors, but such well designed types cannot be put into unions. We don't see a compelling reason to prohibit this.

For example, given this simple little point type:

```
struct point {
   point() {}
   point(int x, int y) : x_(x), y_(y) {}
   int x_, y_;
};
```

This could be a very useful class if space is a problem:

```
class widget {
public:

    widget(point p)
    : type_(POINT)
    , p_(p)
    {}

    widget(int x, int y)
    : type_(POINT)
    , p_(x, y)
    {}

    widget(int i)
    : type_(NUMBER)
    , i_(i)
    {}

    widget(const char* s)
    : type_(TEXT)
    , s_(s)
    {}

    widget(const widget& w)
    {
        *this = w;
    }

    widget& operator=(const widget& w)
    {
        switch (type_ = w.type_)
        {
            case POINT:
                new (&p_) point(w.p_);
                break;
            case NUMBER:
                i_ = w.i_;
                break;
            case TEXT:
                s_ = w.s_;
                break;
        }
        return *this;
    }

private:

    enum { POINT, NUMBER, TEXT } type_;

    union {
        point p_;
        int i_;
        const char* s_;
    };
};
```

Unfortunately this class is illegal because point has non-trivial constructors. (Making the union non-anonymous doesn't help—the restrictions apply to all unions.)

Concerns

The first argument against this goes something like: how does the compiler know what gets constructed and copied? The answer is that the compiler *can't* know that. But that's no reason not to let the programmer tell it what to do. The typical case will be an anonymous union in an enclosing class that also has some discriminating information, so the constructors and assignment operators can be written correctly.

The next concern is that this *does* create a situation where you have a non-constructed object sitting around that could be incorrectly used. But this is what unions are like—this proposal does not create that problem. When you use a union, you have to keep track of which member you are using. It's just as bad to access a float after writing an int as it is to mix up user defined types.

And this is hardly a unique hazard in C++. Consider this code:

```
foo* get_me_a_foo();
foo* f = get_me_a_foo();
f->bar();
```

How do we know what we got back from get_me_a_foo? Is it really pointing to a fully constructed foo, or is it pointing to an int? Or to the printer driver? Who knows? Yet we do this kind of thing all the time under the assumption that get_me_a_foo does the right thing because it claims to return a foo* and it was programmed correctly. This is no different from using a member of a union. You assume that whatever member is marked as active has been correctly constructed or assigned because that's how you programmed it.

In fact, this change will *reduce* risk for the programmer. The current work-around to the union limitations is to create a fake union using template programming or casts. These techniques are difficult to get right, confusing, and messy. Solving the problem with unions makes these tricks unnecessary in a number of very useful cases.

Conclusions

For some applications, the space savings provided by unions are a very attractive feature, and unions are almost powerful enough to be really useful. But they are made second-class citizens by the limitations on membership. Removing a few of these limitations makes unions much more powerful and useful at very little cost to implementers and no additional risk (or perhaps reduced risk) to the programmer.

Acknowledgements

This paper is the result of a discussion between the authors that began at the 2007-Toronto meeting. It supercedes Lois's paper N2248 and draws from that paper and from a paper that Alan had sent to Lois but decided not to submit for the pre-Toronto mailing.

Thanks to Beman Dawes and Howard Hinnant for reviewing drafts and making helpful suggestions, and providing a reality check.