Unrestricted Unions (Revision 2)

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

Lois Goldthwaite
Lois@LoisGoldthwaite.com

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

Jens Maurer
jens.maurer@gmx.net

Document number: **N2544=08-0054**
Date: 2008-02-29

Project: Programming Language C++
Reference: N2521=08-0031

Abstract

This paper proposes a 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.

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 limitation seems unnecessary, and without it unions can become very useful tools for solving certain problems.

Many important problem domains require either large numbers of objects or 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. A typical application of this would be to create a heterogeneous collection of unrelated types which are not dynamically allocated (perhaps they are in-place constructed in a map, or members of an array).

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 any 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 widget class could be very useful if space is a problem and a widget will never need to hold more than one member at once:

```
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.)

As another example, this union is also illegal:

```
union fred {
    long l;
    double d;
    complex<double> c;
};
```

even though unions containing complex numbers are legal in C!

Solution

Our proposed solution is to remove all of the restrictions on the types of members of unions, with the exception of reference types. Any non-reference type may be a union member. We have removed the restriction on static members (for non-anonymous unions) as well since there appears to be no reason for this restriction, and static members are as useful for unions as for other class types.

We have also changed the way implicitly declared special member functions of unions are generated in the following way: if a non-trivial special member function is defined for any member of a union, or a member of an anonymous union inside a class, that special member function will be implicitly deleted (8.4 ¶10) for the union or class. This prevents the compiler from trying to write code that it cannot know how to write, and forces the programmer to write that code if it's needed. The fact that the compiler can't write such a function is no reason not to let the programmer do so.

The current standard specifies that unions have *all* their members automatically default-initialized (as in a struct) through the automatic initialization of members not named in the mem-initializer-list (12.6.2 ¶4). This is technically wrong since only one member can be active at once, but does not cause a problem in practice since the constructors of all members must be trivial. Since we are allowing non-trivial members, we need to fix this by suppressing the automatic initialization of members for unions. (The current standard also specifies member-wise copying of trivial unions, but this will be the subject of a separate core issue.) We also suppress the automatic destruction of union members.

An implication of this is that because members are not automatically initialized, if the programmer fails to list the desired active member in the mem-initializer-list, it will not be constructed. This may or may not be an error since the body of the constructor may take care of the member construction, but the compiler cannot help here.

A subtle implication of allowing non-trivial members is that programmers can no longer always use copy assignment to change the active field. Since it is in not in general valid to assign to an unconstructed object, changing fields may require destruction and construction. For example, consider a union u of type U containing a member m of type M and a member n of type N. If M has a non-trivial destructor and N has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member must be switched from m to n using the destructor and placement new operator. So the N assignment operator might look like this:

```
U& U::operator=(const N& new_n)
{
    m.~M();
    new (&n) N(new_n);
    return *this;
}
```

Concerns

The first concern is that this *does* create a situation where an unconstructed field could be accessed. 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 more complicated types.

Another argument is that the programmer could write the (potentially tricky) special member functions incorrectly. But again, unions are like this, and this is hardly a unique hazard in C++ (or many other languages). Consider this simple 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.

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. Removing the limitations on unions makes these tricks unnecessary.

Proposed Wording

The existing wording of the pertinent sections of 12.1 and 12.4 is not symmetrical, although the meaning is the same. Perhaps there should be a separate issue to address making all the wording consistent in all the sections mentioned below.

7.1.7 Alignment specifier [dcl.align]

10 ~~[*Note:* the `std::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*]~~

8.4 Function definitions [dcl.fct.def]

9 A function definition of the form:

decl-specifier-seq$_{opt}$ *declarator* `= default ;`

is called an *explicitly-defaulted* definition. Only special member functions may be explicitly defaulted, and the implementation shall define them as if they had implicit definitions (12.1, 12.4, 12.8). A special member function that would be implicitly defined as deleted shall not be explicitly defaulted. A special member function is *user-provided* if it is user-declared and not explicitly defaulted on its first declaration. A user-provided explicitly-defaulted function is defined at the point where it is explicitly defaulted. [*Note:* while an implicitly-declared special member function is inline (clause 12), an explicitly-defaulted definition may be non-inline. Non-inline definitions are user-provided, and hence non-trivial (12.1, 12.4, 12.8). This rule enables efficient execution and concise definition while enabling a stable binary interface to an evolving code base. —*end note*] [Example:

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 (clause 9) shall not be a member of a union, nor shall an array of such objects.~~ If a union contains a non-static data member ~~or a member~~ of reference type the program is ill-formed. [*Note:* If any non-static data member of a union has a non-trivial default constructor (12.1 [class.ctor]), copy constructor (12.8 [class.copy]), copy assignment operator (12.8 [class.copy]), or destructor (12.4 [class.dtor]), the corresponding member function of the union must be user-declared or it will be implicitly deleted (8.4 [dcl.fct.def]) for the union. —*end note*]

[*Example:* Consider the following union:

```
union U {
    int i;
    float f;
    std::string s;
};
```

Since std::string (21.2 [string.classes]) declares non-trivial versions of all of the special member functions, U will have an implicitly deleted default constructor, copy constructor, copy assignment operator, and destructor. To use U, some or all of these member functions must be user-declared.

Consider a union u of type U containing non-static data members m of type M and n of type N. If M has a non-trivial destructor and N has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member can be safely switched from m to n using the destructor and placement new operator as follows:

```
u.m.~M();
new (&u.n) N;
```

—*end example*]

Add a new paragraph at the end of section 9.5:

5 A *union-like class* is either a union or a class that has an anonymous union as a direct member. A union-like class X has a set of *variant members;* if X is a union its variant members are the non-static data members, otherwise its variant members are the non-static data members of all anonymous unions that are members of X.

12 Special member functions [special]

1 The default constructor (12.1), copy constructor and copy assignment operator (12.8), and destructor (12.4) are special member functions. [*Note:* The implementation will implicitly declare these member functions for ~~a~~ some class types when the program does not explicitly declare them~~, except as noted in 12.1~~. The implementation will implicitly define them if they are used~~, .~~ as specified in~~See 12.1, 12.4 and 12.8. —*end note*] Programs shall not define implicitly-declared special member functions. Programs may explicitly refer to implicitly declared special member functions. [*Example:* a program may explicitly call, take the address of or form a pointer to member to an implicitly declared special member function.

```
struct A { };                   // implicitly-declared A::operator=
struct B : A {
     B& operator=(const B &);
};
B& B::operator=(const B& s) {
     this->A::operator=(s);   // well-formed
     return *this;
}
```

—*end example*]

12.1 Constructors [class.ctor]

5 A default constructor for a class X is a constructor of class X that can be called without an argument. If there is no user declared constructor for class X, a default constructor is implicitly declared. An implicitly-declared default constructor is an inline public member of its class. If X is a union-like class that has a variant member with a non-trivial default constructor, an implicitly-declared default constructor is defined as deleted (8.4 [dcl.fct.def]).

A default constructor is trivial if it is not user-provided (8.4) and if:

— its class has no virtual functions (10.3) and no virtual base classes (10.1), and

— all the direct base classes of its class have trivial default constructors, and

— for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

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

12.4 Destructors [class.dtor]

3 If a class has no user-declared destructor, a destructor is declared implicitly. An implicitly-declared destructor is an inline public member of its class. If the class is a union-like class that has a variant member with a non-trivial destructor, an implicitly-declared destructor is defined as deleted (8.4 [dcl.fct.def]).

A destructor is trivial if it is not user-provided and if:

— all of the direct base classes of its class have trivial destructors and

— for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

6 After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class X calls the destructors for X's direct non-variant members, the destructors for X's direct base classes and, if X is the type of the most derived class (12.6.2), its destructor calls the destructors for X's virtual base classes. All destructors are called as if they were referenced with a qualified name, that is, ignoring any possible virtual overriding destructors in more derived classes. Bases and members are destroyed in the reverse order of the completion of their constructor (see 12.6.2). A return statement (6.6.3) in a destructor might not directly return to the caller; before transferring control to the caller, the destructors for the members and bases are called. Destructors for elements of an array are called in reverse order of their construction (see 12.6).

~~9 A union member shall not be of a class type (or array thereof) that has a non-trivial destructor.~~

12.6.2 Initializing bases and members [class.base.init]

4 If a given non-static data member or base class is not named by a mem-initializer-id (including the case where there is no mem-initializer-list because the constructor has no ctor-initializer), then

— If the entity is a non-static non-variant data member of (possibly cv-qualified) class type (or array thereof) or a base class, and the entity class is a non-trivial class, the entity is default-initialized (8.5). If the entity is a non-static data member of a const-qualified type, the entity class shall have a user-provided default constructor.

— Otherwise, the entity is not initialized. If the entity is of const-qualified type or reference type, or of a (possibly cv-qualified) trivial class type (or array thereof) containing (directly or indirectly) a member of a const-qualified type, the program is ill-formed.

After the call to a constructor for class X has completed, if a member of X is neither specified in the constructor's meminitializers, nor default-initialized, nor value-initialized, nor given a value during execution of the compound-statement of the body of the constructor, the member has indeterminate value.

12.8 Copying class objects [class.copy]

4 If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class is a union-like class that has a variant member with a non-trivial copy constructor, an implicitly-declared copy constructor is defined as deleted (8.4 [dcl.fct.def]).

Thus, for the class definition

```
struct X {
     X(const X&, int);
};
```

a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of X's copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

10 If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly. If the class is a union-like class that has a variant member with a non-trivial copy assignment operator, an implicitly-declared copy assignment operator is defined as deleted (8.4 [dcl.fct.def]).

The implicitly-declared copy assignment operator for a class X will have the form

```
X& X::operator=(const X&)
```

if

— each direct base class B of X has a copy assignment operator whose parameter is of type const B&, const volatile B& or B, and

— for all the non-static data members of X that are of a class type M (or array thereof), each such class type has a copy assignment operator whose parameter is of type const M&, const volatile M& or M.

Otherwise, the implicitly declared copy assignment operator will have the form

```
X& X::operator=(X&)
```

15.2 Constructors and destructors [except.ctor]

2 An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed ~~subobjects~~base classes and non-variant members, that is, for subobjects for which the principal constructor (12.6.2) has completed execution and the destructor has not yet begun execution. Similarly, if the non-delegating constructor for an object has completed execution and a delegating constructor for that object exits with an exception, the object's destructor will be invoked. Should a constructor for an element of an automatic array throw an exception, only the constructed elements of that array will be destroyed. If the object or array was allocated in a new-expression, the matching deallocation function (3.7.3.2, 5.3.4, 12.5), if any, is called to free the storage occupied by the object.

15.3 Handling an exception [except.handle]

11 The fully constructed base classes and members of an object shall be destroyed before entering the handler of a function try-block of a constructor ~~or destructor~~ for that object. Similarly, if a delegating constructor for an object exits with an exception after the non-delegating constructor for that object has completed execution, the object's destructor shall be executed before entering the handler of a function-try-block of a constructor for that object. The base classes and non-variant members of an object shall be destroyed before entering the handler of a function try-block of a destructor for that object (12.4 [class.dtor]).

Conclusions

For some applications, the space savings provided by unions are a very attractive feature, and unions have the potential to be really useful. But they are made second-class citizens by the limitations on membership. Removing 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 and continued at the 2007-Kona 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.

Thanks to Alisdair Meredith and Bjarne Stroustrup for pointing out problems with the first version of the paper.

Thanks to Pete Becker for checking some language and making helpful suggestions.

Revision History

Revision 1 - Since N2412

Added Solution section. Removed restriction on non-trivial destructors. Removed call to non-trivial default constructor of first-named member. Added language to indicate that implicit members will not be declared if members have non-trivial versions. Added language to suppress the automatic initialization and destruction of members. Added lifting of static member restriction. Made numerous other editorial changes.

Revision 2 - Since N2430

Made changes throughout to delete special member functions rather than simply fail to declare them. Added the 8.4, 15.2 and 15.3 sections. Made a number of changes per direction from the Core Working Group.