Type Issues and Proposed Resolutions
====================================

621 - Is a definition for the terms "same type" needed?

Does the WP need to define what it means for two objects/expressions to have the same type? I need help (i.e. inspiration) as to how we would go about doing this...

Looking through the WP where the terms "same type" is used, I noticed the following problems:

- 8.5.1 [dcl.init.aggr] para 15
  "The initializer for a union with no user-declared constructor is either a single expression of the _same type_, or a brace-enclosed initializer for the first member of the union."
  This should say:
  "...the same type (ignoring the top-level cv-qualifiers)"

- 12.8 [class.copy] para 15
  "Whenever a class object is copied and the original object and the copy have the _same type_, if the implementation can prove that either the original object or the copy will never again be used except as the result of an implicit destructor call (_class.dtor_), an implementation is permitted to treat the original and the copy as two different ways of referring to the same object and not perform a copy at all."

  This should say:
  "...the same type (ignoring the top-level cv-qualifiers)"

- 15.3 [except.handle] para 2
  "A handler with type T, const T, T&, or const T& is a match for a throw-expression with an object of type E if -- T and E are the same type, ..."

  This should say:
  "...the same type (ignoring the top-level cv-qualifiers of type E)"

213 - Should vacuous type declarations be prohibited?

7 [dcl.dcl] para 1 says:
"A declaration introduces one or more names into a program and specifies how those names are to be interpreted."

Is this intended to prohibit empty declarations like these?

```c
enum { };
class { int i; };
class { };
typedef enum {};
```

In this case the WP should be clearer.

Jerry Schwarz also noted:
> This can also be interpreted as prohibiting the following:
> extern int i;
> extern int i;
> since the second declaration does not introduce anything (the name has already been introduced in the program).
Proposal:

I do not have a strong preference for this...
I decided that saying what the C standard says was a safe thing.
Vacuous declarations are ill-formed.
Rewrite 7[dcl.dcl] para 1 as follows:

"A declaration shall introduce one or more names into a program, or shall redeclare a name introduced by a previous declaration. A declaration specifies how those names are to be interpreted."

116 - Is "const class X { }" legal?

Mike Miller asks the following:
> Is "const class X { }" legal, and, if so, what does it mean?
> If the declaration does not declare a declarator and a storage class specifier or a cv-qualifier is specified, are these simply ignored or is the declaration ill-formed?

Solution 1):

Add to 7[dcl.dcl], at the end of para 3:

"In these cases, if the decl-specifier-seq contains a cv-qualifier (7.1.5.1, dcl.type.cv) or a storage class specifier (7.1.1, dcl.stc), these specifiers are ignored."

564 - is 'void f(const a);' well-formed?

The working paper says, in 7.1.5[dcl.type] para 3:

"At least one type-specifier is required in a typedef declaration. At least one type-specifier is required in a function declaration unless it declares a constructor, destructor or type conversion operator.56)
56) There is no special provision for a decl-specifier-seq that lacks a type-specifier. The "implicit int" rule of C is no longer supported."

Annex C gives the following example:

"void f(const parm); // invalid C++"

A cv-qualifier (like const in the example above) is a type-specifier. So, according to the rule above, the example is valid, i.e. a declaration that has only cv-qualifiers in its type-specifier is valid according to 7.1.5.

Is the rule in 7.1.5 incorrect or is the example incorrect?

Proposal:

The example above is ill-formed.

Change in 7.1.5[dcl.type] paragraph 3 to say:

"At least one type-specifier that is not a cv-qualifier is required in
a typedef declaration. At least one type-specifier that is not a

cv-qualifier is required in a function declaration unless it declares

a constructor, destructor or type conversion operator.56)

56) There is no special provision for a decl-specifier-seq that

lacks a type-specifier or that has a type-specifier that only

specifies cv-qualifiers. The "implicit int" rule of C is no

longer supported."

503 - Clarifications for bitfields of enumeration type needed

Question 1):

----------

Bill Gibbons mentionned:

> 7.2[dcl.enum] paragraph 5 describes the underlying type of

> enumeration types. It should be made clear that this description

> does not apply to the underlying type of enumeration bit-fields.

Proposal:

=========

Change the beginning of 7.2 paragraph 5 to say:

"The underlying type of an enumeration FN)...  

-----

FN) This does not apply to the underlying type of bitfields of

enumeration type."

Question 2):

----------

Bill Gibbons mentionned:

> Also, something should be said about the signedness of enumeration

> types. Suggested new words:

> "Even though the underlying type of an enumeration will be either

>  signed or unsigned, enumerations themselves are neither signed

>  nor unsigned. [For example, a two-bit bit-field can hold an

>  enumeration with values {0,1,2,3}.]

Proposal:

=========

Add the words Bill suggests at the end of 7.2 paragraph 5.

47 - bitfields & number of bits required by its type

Question 1:

----------

Can a bit-field be declared with less bits than what is required to

store all of the values of its type?

    enum ee { one, two, three, four };
    struct S {
      ee  bit1:1; // well-formed?
    };

Solution 1)

----------

The declaration is ill-formed.

The number of bits of a bit-field of enumeration type shall be

sufficient to hold all of the values of the enumeration type.

Solution 2)

----------

The declaration is well-formed.

Since, for all other bit-field types (beside enumeration), a

bit-field can be declared with less bits than what is necessary to

hold all of the values of its type, bit-fields of enumeration type

should not be different.

Proposal:
I slightly prefer 2). I could live with both solution.

Question 2: 
--------------
struct S {
    char bit2:16;// well-formed?
};

Proposal: 
--------------
The declaration is ill-formed. The number of bits in a bit-field declaration shall not be greater than the number of bits needed for the object representation of the bit-field’s type, or if the bitfield is of enumeration type, of the enumeration’s underlying type.

623 - Representation of bitfields of bool type

9.6[class.bit] paragraph 3 says:
"A bool value can be successfully stored in a bit-field of any nonzero size."
What does it mean "can be successfully stored"?

Proposal:
--------------
Replace the sentence above with:
"If a bool value is stored into a bit-field of type bool of any nonzero size (including a one-bit bitfield), the value of the bit-field and the original bool value shall be the same."

458 - When is an enum bitfield signed / unsigned?

Sam Kendall noted:
> enum Bool { false=0, true=1 };  
> struct A {  
>    Bool b:1;  
> };  
> A a;  
> a.b = true;  
> if (a.b == true) // if this is sign-extended, this fails.

Proposal:
--------------
Bill Gibbons proposed the following resolution:
After the sentence 9.6[class.bit] paragraph 3, at the end of the 2nd sentence:
"It is implementation defined whether plain (neither explicitly signed or unsigned) char, wchar_t, short, int or long bitfield is signed or unsigned."
add the following:
"...; bit-fields of enumeration type are neither signed nor unsigned. [For example, a two-bit bit-field can hold an enumeration with values {0,1,2,3}.]"

571 - Is bitfield part of the type?

Bill Gibbons mentioned:
> The description in 4.5 [conv.prom] para 3 seems to indicate that bitfield is part of the type. Is it?
>
> If it is (as 4.5 seems to indicate) this subclause should be more explicit about it. If it isn’t, bitfields should be discussed in lvalue/rvalue subclause [basic.lval] to describe how a bitfield lvalue is transformed into an rvalue.
Proposal:
==========
No, the bit-field attribute is not part of the type.
Add to 4.1[conv.lval] at the end of paragraph 1:
"If the lvalue refers to a bitfield of type T, the resulting
rvalue is not a bitfield."

267 - What does "Nor are there any references to bitfields" mean?

9.6[class.bit] paragraph 3 says:
"Nor are there references to bit-fields."

Tom Plum & Dan Saks ask the following:
> Does this actually prohibit anything? A simple attempt to make a
> reference refer to a bit-field just creates a temporary:
> union { int bitf:2; } u;
> const int & r = u.bitf;
> Or is this a syntactic restriction that prohibits something like
> union { int (&rbitf):2 } u;
> Or is it meant to prohibit the use of typedefs to attempt it, such as
> union { typedef int bitf_t:2; bitf_t &rbitf; } u;
> The intent needs clarifying.

Proposal:
==========
Replace the sentence above with:
"A reference shall not be initialized with an lvalue that
represents a bit-field."

568 - Can a POD class have a static member of type pointer-to-member,
non-POD-struct or non-POD-union?

9 [class] paragraph 4 says:
"A POD-struct is an aggregate class that has no members of type
pointer-to-member, non-POD-struct or non-POD-union (or arrays of
such types) or reference, and has no user-defined copy assignment
operator and no user-defined destructor."
And similar wording for POD-union.

An aggregate can have static members.
The wording above allows a POD class to have static members as well.
However, it prohibits static members of type "pointer-to-member,
non-POD-struct or non-POD-union (or arrays of such types) or
reference". Should it?

Proposal:
==========
I don’t see why it should.
The sentence above should say:
"A POD-struct is an aggregate class that has no _non-static_
members ...."
and similarly for POD-union.