Proposal of Bit-field Default Member Initializers

Document No.: Dnnnn
Project: Programming Language C++ - Evolution
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2015-11-20

Summary

We propose default member initializers for bit-fields.

Example:

```
struct S {
  int x : 6 = 42;
};
```

To ease parsing we specify a rule, roughly summarized as “you have to use the =, and the = always starts the initializer”.

Background

The declarators of class members are called `member-declarators`:

```
member-declarator:
    declarator virt-specifier-seq_opt pure-specifier_opt
    declarator brace-or-equal-initializer_opt
    identifier_opt attribute-specifier-seq_opt: constant-expression
```

As can be seen, non-bit-field members may have default member initializers. Bit-fields may not.

The motivation for having initializers for bit-fields is the same as having initializers for non-bit-fields. It can be argued that the motivation is even stronger for bit-fields, as they usually occur in “simple structs” where member initializers are heavily used for their tersity/compactness.

Naively adding them...

```
member-declarator:
```

```
declarator virt-specifier-seqopt pure-specifier_opt
declarator brace-or-equal-initializer_opt
identifier_opt attribute-specifier-seq_opt: constant-expression \
  brace-or-equal-initializer_opt
```

...creates parsing difficulties and parsing ambiguities. In particular, if a constant-expression is immediately followed by an optional brace-or-equal-initializer, it can be unclear if a non-nested `=` or `{` is the first token of the initializer or a continuation of the constant-expression, and in some of those cases this remains ambiguous even with infinite lookahead.

Proposal

We propose adding the initializer to the grammar as per the above and then adding a couple of special parsing rules that serves to both (a) resolve potential ambiguities; and (b) make it easy to parse.

Roughly, the first proposed rule is that, in a bitfield declarator, the first non-nested = token terminates parsing of the constant-expression.

Consequences: A bitfield width may not contain a non-nested = token. A non-nested = token after the : token in a bitfield declarator unambiguously commences the initializer in a well-formed program.

Rationale: It would be a very strange constant-expression that uses an overloaded assignment operator. In such bizarre cases, it remains possible to wrap the bitfield width in parenthesis to get it to parse as intended.

Roughly, the second proposed rule is that, in a bitfield declarator, a { token does not start parsing of the brace-or-equal-initializer.

Consequences: The initializer of a bitfield must start with an = token. That is, it must use the copy-initialization or copy-list-initialization form, and may not use the direct-initialization or direct-list-initialization form. Informally the rule is “you have to use the equals” in a bitfield default member initializer.

Rationale: For a bit-field, there is no difference between copy-initialization and direct-initialization (likewise no difference between copy-list-initialization and direct-list-initialization). Therefore a would-be use of the direct forms can be replaced with the copy forms, without semantic difference. For this reason, we resolve the opening brace to the constant expression.

Wording

Add to member-declarator:

```
member-declarator:
    declarator virt-specifier-seqopt pure-specifier_opt
    declarator brace-or-equal-initializer_opt
    identifier_opt attribute-specifier-seq_opt: constant-expression \
      brace-or-equal-initializer_opt
```

Add to [class.bit]:

A member-declarator of the form:

```
    identifier_opt attribute-specifier-seq_opt: constant-expression \
      brace-or-equal-initializer_opt
```

New paragraph in [class.bit]:

During parsing of the constant-expression in a bitfield member-declarator:

- Non-nested { tokens are taken as part of the constant-expression. [Note: Such tokens are never taken as the start of the following brace-or-equal-initializer.]
- A non-nested = token is not taken as part of the constant-expression. [Note: The token is taken as the start of the following brace-or-equal-initializer.]

[Example:

```
  struct S {
    int a :
    b ? c : d // constant-expression
    = e; // brace-or-equal-initializer

    int x :
    y { z } // constant-expression
    ; // no brace-or-equal-initializer
  }
```

]