Proposal of Bit-field Default Member Initializers

Document No.: P0187R0
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 =, and = always starts the initializer". We apply this rule by adjusting the grammar.

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.

For example:

```
struct S {
  int y : true ? 1 : a = 42; // Is 42 a default member initializer
                             // or the rhs of an assignment?
  int x : 1 || new int { 43 }; // Is 43 a default member initializer
                               // or part of the new expression?
};
```

Proposal

We propose to resolve these ambiguities by effectively 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:

1. For a bit-field, there is no difference in effect 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.
2. Leaving it ambiguous lead to complaints about implementation difficulty.
3. Non-nested braces are useful in constant expressions. For example:

```
enum E { k = 4; };
struct X { int n : int{E::k}; };
```

Weighing these three points we decided to propose that the brace be given to the constant expression.

We apply these two rules by adjusting the grammar, reducing the would-be *constant-expression* to not allow non-nested = syntactically, and reducing the would-be *brace-or-equal-initializer* to = *initializer*.

Wording

Add to grammar and member-declarator:

member-declarator:
    declarator virt-specifier-seq$_{opt}$ pure-specifier$_{opt}$
    declarator brace-or-equal-initializer$_{opt}$
    *identifier$_{opt}$ attribute-specifier-seq$_{opt}$: *
      noassign-conditional-expression
    *identifier attribute-specifier-seq$_{opt}$: *
      noassign-conditional-expression = initializer

noassign-conditional-expression :
    logical-or-expression
    *logical-or-expression ? noassign-conditional-expression : *
      noassign-conditional-expression

Modify [class.bit]:

A *member-declarator* of one of the forms:

    *identifier$_{opt}$ attribute-specifier-seq$_{opt}$: *

```
    noassign-conditional-expression
  identifier attribute-specifier-seq_opt : \
    noassign-conditional-expression = initializer
```

specifies a bit-field; its length is set off from the bit-field name by a colon. If an = *initializer* is present, it is treated as a *brace-or-equal-initializer* of this data member ([class.mem]/4). The optional *attribute-specifier-seq* appertains to the entity being declared. The bit-field attribute is not part of the type of the class member. The *`noassign-conditional-expression`* shall be an integral constant expression with a value greater than or equal to zero.

Add new paragraph to [class.bit]:

A *`noassign-conditional-expression`* is equivalent, by definition, to a *`conditional-expression`* that consists of the same sequence of tokens.

Add new section [diff.cpp14.class]:

Change: Change bit-field widths to be *noassign-conditional-expressions*.
Rationale: To enable bit-field default member initializers.
Effect on original feature: Valid C++ 2014 code may fail to compile or change meaning in this International Standard:

```
  int a;
  struct S {
    int b : true ? 2 : a = 1;
    // before: bit-field width of 2, not initialized
    // after: bit-field width of 2, initialized with the value 1.
  };
```

References

CWG ISSUE 1341
Thread starting [c++std-core-28391] “another observation on core issue 1341”
CWG Kona 2015 discussion about same

Acknowledgements

Thanks to James Widman for submitting the original core issue asking for this feature. Thanks to Richard Smith for the initial discussion that lead to this proposal. Thanks to Jens Maurer for reviewing and presenting this proposal.