Document Number P0329R0

Date 2016-05-09

Authors Tim Shen <timshen@google.com>
Richard Smith <richard@metafoo.co.uk>
Zhihao Yuan <zy@miator.net>
Chandler Carruth <chandlerc@google.com>

Audience EWG

Designated Initialization

Introduction

This proposal introduces a new syntax based on C99 designated initializers to initialize an aggregate:

```
Point p{ .x = 3.0, .y = 4.0 };
distance({ .x{}, .y{5.0} }, p);  // direct-list-initialization
```

by specifying pairs of public data member *designators* followed by *brace-or-equal-initializers*.

Motivation

To increase readability and explicitness: With initializations designated with data member names, the code states its intent more explicitly and avoids the possibility of bugs due to initializers being matched against the wrong member.

Towards more flexible and sustainable aggregate initialization: Compared to list-initialization, designated initialization allows the user to enumerate only the interesting data members, leaving the rest default member initialized. In this way, the initialization is less sensitive to data member changes.

To increase the interoperability between C and C++: By being compatible with C designated initialization, C++ is more interoperable with C code, e.g. easy initialization of a C struct, and allowing designated initialization code in a header file that may be accessed by both C and C++ compilers.

To standardize and generalize the existing practices: Clang already implements a designated initializer extension that is C-compatible. GCC implements “trivial designated initializer”, which requires the designators to appear in the same order as declarations.

Design Decisions

The proposed C++ designated initialization is based on C99 designated initializers, with following differences:

C99	C++ with this proposal	Example
Designators are optional	Either all designators, or none	`C only:` `A a = { 3, .a = 4 }`

Initialized from left to right. Evaluation order is unspecified	Designators must appear in the declaration order of the data members. Evaluation order is left to right.	`struct { int a, b; };`
		
		`A a = {.b = 3, .a = 4}`
		`C: .b = 3, then .a = 4`
		`C++: Error - .a must come first`

Designators may be duplicated	Designators are required to be unique	`C only:` `A a = { .a = 3, .a = 4 }`
Supports array designated initialization	Does not support array designated initialization	`C only:` `A a = { [3] = 4 }`
Designators can be nested	Designators cannot be nested	`C only:` `A a = { .e.a = 3 }`
Supports C initializer	Supports C++ brace-or-equal-initializer	`C++ only:` `A a = { .a{} }`

Evaluation and Initialization Orders

Notice that there are two kinds of orders:

- The order to evaluate the initializers
- The order to perform the actual initializations

To be consistent with list-initialization, we expect the initializers to be evaluated in left-to-right order, as written; but we also want to perform the actual initializations in data members' declaration order, so that they can be destructed in the reverse order. When these two orders

do not match, an implementation cannot avoid creating temporaries to fill the gap between the evaluations and initializations.

To meet these expectations for guaranteed copy elision, we require the designators to appear as a subsequence of the data member declaration sequence, so that the evaluation order matches the declaration order, and it is also textually left-to-right in designated initialization.

Base Class Object Initialization

The base class objects will be initialized with {}. We do not have a concrete use case for some in-depth control of how to initialize the base class objects, and the proposed design is forward compatible, therefore we suggest to address this issue in another proposal.

Proposed Design

Syntax

initializer:
    brace-or-equal-initializer
    (expression-list)
brace-or-equal-initializer:
    = initializer-clause
    braced-init-list
initializer-clause:
    assignment-expression
    braced-init-list
initializer-list:
    initializer-clause ...$_{opt}$
    initializer-list , initializer-clause ...$_{opt}$
braced-init-list:
    { initializer-list ,$_{opt}$ }
    { designated-initializer-list ,$_{opt}$ }
    { }
designated-initializer-list:
    designated-initializer-clause
    designated-initializer-list , designated-initializer-clause
designated-initializer-clause:
    designator brace-or-equal-initializer
designator:
    . identifier

Technical Specification

If T is a non-array aggregate type, a designated initialization of an object with type T may be performed, with following requirements:

- The *identifier* in a designator should be the name of a non-static data member of T.
- Each non-static data member should be designated for at most once.
- Two data members that are part of the same union, for any possible union in T or T itself, should not be both designated.
- All designators must appear as a subsequence of the data member declarations.

Notice that the last requirement only takes place for initialization. In other cases, e.g. overload resolution, the designators are still considered unordered. Thus, in overload resolution, a candidate is viable if it satisfies the above requirements *except* the last one.

A *braced-init-list* with designators as a function argument causes the parameter to be considered a non-deduced context.

Example:

```
struct A {
    int a, b;
};
struct B {
    int b, a;
};
struct C {
    int a, c;
};
void Foo(A);
void Foo(B);
void Bar(B);
void Bar(C);

int main() {
    Foo({ .a = 3, .b = 4 });  // Ambiguous: Foo(A) or Foo(B)?
    Bar({ .a = 3, .b = 4 });  // Error: Resolve to Bar(B), but
                              // designators are in the wrong order.
    Bar({ .a = 3, .c = 4 });  // Resolves to void Bar(C).
    Bar({ .a = 3 });          // Ambiguous: Bar(B) or Bar(C)?
}
```

Initialization is formed in following rules:

- Base class objects are initialized with {}.
- For each designated data member, initialize it with the *brace-or-equal-initializer* that comes after the corresponding designator.
- For each non-static *direct* data member that is not specified by the above step, initialize it with its default member initializer, if present, otherwise {}.
- All initializations are performed in declaration order.

Example:

```
struct B {
    int base_b;
};
struct A : B {
    int a;
    int b = 3;
    union {
        union {
            int u0;
            int u1;
        };
        int u2;
    };
    union {
        int u4;
        int u5;
    };
    union C {
        int c1;
        int c2 = 7;
    } c;
    union D {
        int d1;
        int d2;
    } d;
    int e;
    string f;
};
```

The designated initialization code:

```
{ .a = 42, .u1 = 6, .d { .d2 = 2 }, .f{"a"}, }
```

has the following steps:

1. Initialize base B object with {}.
2. Initialize a with = 42;
3. Initialize b with = 3;
4. Initialize u1 with = 6;
5. Initialize the anonymous union { int u4; int u5; } with {}, which means initialize u4 with {};
6. Initialize c with {};
7. Initialize d with { .d2 = 2 };
8. Initialize e with {};
9. Initialize f with {"a"}.

Future Issues

- Do we allow the constructor initializer syntax, like `{ .name('x', 4) }` ?
- Do we allow a designation list to appear in a list-initializer as a suffix, e.g. `A { 1, 2, .c = 3, .d = 4 }` ?

References

1. GCC Designated Initializers: https://gcc.gnu.org/onlinedocs/gcc/Designated-Inits.html
2. Clang implementation: https://github.com/llvm-mirror/clang/blob/master/include/clang/AST/Expr.h#L3935