13.9 — Default member initialization

When we define a struct (or class) type, we can provide a default initialization value for each member as part of the type definition. This process is called **non-static member initialization**, and the initialization value is called a **default member initializer**.

Here's an example:

```
struct Something
{
    int x;       // no initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s1; // s1.x is uninitialized, s1.y is 0, and s1.z is 2

    return 0;
}
```

In the above definition of `Something`, `x` has no default value, `y` is value-initialized by default, and `z` has the default value `2`. These default member initialization values will be used if the user doesn't provide an explicit initialization value when instantiating an object of type `Something`.

Our `s1` object doesn't have an initializer, so the members of `s1` are initialized to their default values. `s1.x` has no default initializer, so it remains uninitialized. `s1.y` is value initialized by default, so it gets value `0`. And `s1.z` is initialized with the value `2`.

Note that even though we haven't provided an explicit initializer for `s1.z`, it is initialized to a non-zero value because of the default member initializer provided.

Key insight

Using default member initializers (or other mechanisms that we'll cover later), structs and classes can self-initialize even when no explicit initializers are provided!

For advanced readers

CTAD (which we cover in lesson 13.14 -- Class template argument deduction (CTAD) and deduction guides) cannot be used in non-static member initialization.

Explicit initialization values take precedence over default values

Explicit values in a list initializer always take precedence over default member initialization values.

```
struct Something
{
    int x;       // no default initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s2 { 5, 6, 7 }; // use explicit initializers for s2.x, s2.y, and s2.z
(no default values are used)

    return 0;
}
```

In the above case, `s2` has explicit initialization values for every member, so the default member initialization values are not used at all. This means `s2.x`, `s2.y` and `s2.z` are initialized to the values `5`, `6`, and `7` respectively.

Missing initializers in an initializer list when default values exist

In the previous lesson (13.8 -- Struct aggregate initialization) we noted that if an aggregate is initialized but the number of initialization values is fewer than the number of members, then all remaining members will be value-initialized. However, if a default member initializer is provided for a given member, that default member initializer will be used instead.

```
struct Something
{
    int x;       // no default initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s3 {}; // value initialize s3.x, use default values for s3.y and s3.z

    return 0;
}
```

In the above case, `s3` is list initialized with an empty list, so all initializers are missing. This means that a default member initializer will be used if it exists, and value initialization will occur otherwise. Thus, `s3.x` (which has no default member initializer) is value initialized to `0`, `s3.y` is value initialized by default to `0`, and `s3.z` is defaulted to value `2`.

Recapping the initialization possibilities

If an aggregate is defined with an initialization list:

- If an explicit initialization value exists, that explicit value is used.
- If an initializer is missing and a default member initializer exists, the default is used.
- If an initializer is missing and no default member initializer exists, value initialization occurs.

If an aggregate is defined with no initialization list:

- If a default member initializer exists, the default is used.
- If no default member initializer exists, the member remains uninitialized.

Members are always initialized in the order of declaration.

The following example recaps all possibilities:

```
struct Something
{
    int x;       // no default initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s1;             // No initializer list: s1.x is uninitialized, s1.y and
s1.z use defaults
    Something s2 { 5, 6, 7 }; // Explicit initializers: s2.x, s2.y, and s2.z use
explicit values (no default values are used)
    Something s3 {};          // Missing initializers: s3.x is value initialized,
s3.y and s3.z use defaults

    return 0;
}
```

The case we want to watch out for is `s1.x`. Because `s1` has no initializer list and `x` has no default member initializer, `s1.x` remains uninitialized (which is bad, since we should always initialize our variables).

Always provide default values for your members

To avoid the possibility of uninitialized members, simply ensure that each member has a default value (either an explicit default value, or an empty pair of braces). That way, our members will be initialized with some value regardless of whether we provide an initializer list or not.

Consider the following struct, which has all members defaulted:

```
struct Fraction
{
	int numerator { }; // we should use { 0 } here, but for the sake of example
we'll use value initialization instead
	int denominator { 1 };
};

int main()
{
	Fraction f1;          // f1.numerator value initialized to 0, f1.denominator
defaulted to 1
	Fraction f2 {};       // f2.numerator value initialized to 0, f2.denominator
defaulted to 1
	Fraction f3 { 6 };    // f3.numerator initialized to 6, f3.denominator
defaulted to 1
	Fraction f4 { 5, 8 }; // f4.numerator initialized to 5, f4.denominator
initialized to 8

	return 0;
}
```

In all cases, our members are initialized with values.

Best practice

Provide a default value for all members. This ensures that your members will be initialized even if the variable definition doesn't include an initializer list.

Default initialization vs value initialization for aggregates

Revisiting two lines from the above example:

```
Fraction f1;          // f1.numerator value initialized to 0, f1.denominator
defaulted to 1
Fraction f2 {};       // f2.numerator value initialized to 0, f2.denominator
defaulted to 1
```

You'll note that `f1` is default initialized and `f2` is value initialized, yet the results are the same (`numerator` is initialized to `0` and `denominator` is initialized to `1`). So which should we prefer?

The value initialization case (`f2`) is safer, because it will ensure any members with no default values are value initialized (and although we should always provide default values for members, this protects against the case where one is missed).

Preferring value initialization has one more benefit -- it's consistent with how we initialize objects of other types. Consistency helps prevent errors.

Best practice

For aggregates, prefer value initialization (with an empty braces initializer) to default initialization (with no braces).

That said, it's not uncommon for programmers to use default initialization instead of value initialization for class types. This is partly for historic reasons (as value initialization wasn't introduced until C++11), and partly because there is a similar case (for non-aggregates) where default initialization can be more efficient (we cover this case in 14.9 -- Introduction to constructors).

Therefore, we won't be militant about enforcing use of value initialization for structs and classes in these tutorials, but we do strongly recommend it.