13.8 — Struct aggregate initialization

In the previous lesson (13.7 -- Introduction to structs, members, and member selection), we talked about how to define structs, instantiate struct objects, and access their members. In this lesson, we'll discuss how structs are initialized.

Data members are not initialized by default

Much like normal variables, data members are not initialized by default. Consider the following struct:

```
#include <iostream>

struct Employee
{
    int id; // note: no initializer here
    int age;
    double wage;
};

int main()
{
    Employee joe; // note: no initializer here either
    std::cout << joe.id << '\n';

    return 0;
}
```

Because we have not provided any initializers, when `joe` is instantiated, `joe.id`, `joe.age`, and `joe.wage` will all be uninitialized. We will then get undefined behavior when we try to print the value of `joe.id`.

However, before we show you how to initialize a struct, let's take a short detour.

What is an aggregate?

In general programming, an **aggregate data type** (also called an **aggregate**) is any type that can contain multiple data members. Some types of aggregates allow members to have different types (e.g. structs), while others require that all members must be of a single type (e.g. arrays).

In C++, the definition of an aggregate is narrower and quite a bit more complicated.

Author's note

In this tutorial series, when we use the term “aggregate” (or “non-aggregate”) we will mean the C++ definition of aggregate.

For advanced readers

To simplify a bit, an aggregate in C++ is either a C-style array (17.7 -- Introduction to C-style arrays), or a class type (struct, class, or union) that has:

- No user-declared constructors (14.9 -- Introduction to constructors)
- No private or protected non-static data members (14.5 -- Public and private members and access specifiers)
- No virtual functions (25.2 -- Virtual functions and polymorphism)

The popular type `std::array` (17.1 -- Introduction to std::array) is also an aggregate.

You can find the precise definition of a C++ aggregate here.

The key thing to understand at this point is that structs with only data members are aggregates.

Aggregate initialization of a struct

Because a normal variable can only hold a single value, we only need to provide a single initializer:

```
int x { 5 };
```

However, a struct can have multiple members:

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};
```

When we define an object with a struct type, we need some way to initialize multiple members at initialization time:

```
Employee joe; // how do we initialize joe.id, joe.age, and joe.wage?
```

Aggregates use a form of initialization called **aggregate initialization**, which allows us to directly initialize the members of aggregates. To do this, we provide an **initializer list** as an initializer, which is just a braced list of comma-separated values.

There are 2 primary forms of aggregate initialization:

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee frank = { 1, 32, 60000.0 }; // copy-list initialization using braced
list
    Employee joe { 2, 28, 45000.0 };     // list initialization using braced list
(preferred)

    return 0;
}
```

Each of these initialization forms does a **memberwise initialization**, which means each member in the struct is initialized in the order of declaration. Thus, `Employee joe { 2, 28, 45000.0 };` first initializes `joe.id` with value `2`, then `joe.age` with value `28`, and `joe.wage` with value `45000.0` last.

Best practice

Prefer the (non-copy) braced list form when initializing aggregates.

In C++20, we can also initialize (some) aggregates using a parenthesized list of values:

```
Employee robert ( 3, 45, 62500.0 );  // direct initialization using parenthesized
list (C++20)
```

We recommend avoiding this last form as much as possible, as it doesn't currently work with aggregates that utilize brace elision (notably `std::array`).

Missing initializers in an initializer list

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.

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 2, 28 }; // joe.wage will be value-initialized to 0.0

    return 0;
}
```

In the above example, `joe.id` will be initialized with value `2`, `joe.age` will be initialized with value `28`, and because `joe.wage` wasn't given an explicit initializer, it will be value-initialized to `0.0`.

This means we can use an empty initialization list to value-initialize all members of the struct:

```
Employee joe {}; // value-initialize all members
```

Const structs

Variables of a struct type can be const (or constexpr), and just like all const variables, they must be initialized.

```
struct Rectangle
{
    double length {};
    double width {};
};

int main()
{
    const Rectangle unit { 1.0, 1.0 };
    const Rectangle zero { }; // value-initialize all members

    return 0;
}
```

Designated initializers C++20

When initializing a struct from a list of values, the initializers are applied to the members in order of declaration.

```
struct Foo
{
    int a {};
    int c {};
};

int main()
{
    Foo f { 1, 3 }; // f.a = 1, f.c = 3

    return 0;
}
```

Now consider what would happen if you were to update this struct definition to add a new member that is not the last member:

```
struct Foo
{
    int a {};
    int b {}; // just added
    int c {};
};

int main()
{
    Foo f { 1, 3 }; // now, f.a = 1, f.b = 3, f.c = 0

    return 0;
}
```

Now all your initialization values have shifted, and worse, the compiler may not detect this as an error (after all, the syntax is still valid).

To help avoid this, C++20 adds a new way to initialize struct members called **designated initializers**. Designated initializers allow you to explicitly define which initialization values map to which members. The members can be initialized using list or copy initialization, and must be initialized in the same order in which they are declared in the struct, otherwise a warning or error will result. Members not designated an initializer will be value initialized.

```
struct Foo
{
    int a{ };
    int b{ };
    int c{ };
};

int main()
{
    Foo f1{ .a{ 1 }, .c{ 3 } }; // ok: f1.a = 1, f1.b = 0 (value initialized), f1.c =
3
    Foo f2{ .a = 1, .c = 3 };   // ok: f2.a = 1, f2.b = 0 (value initialized), f2.c =
3
    Foo f3{ .b{ 2 }, .a{ 1 } }; // error: initialization order does not match order
of declaration in struct

    return 0;
}
```

For Clang users

When doing designated initializers of single values using braces, Clang improperly issues warning “braces around scalar initializer”. Hopefully this will be fixed soon.

Designated initializers are nice because they provide some level of self-documentation and help ensure you don’t inadvertently mix up the order of your initialization values. However, designated initializers also clutter up the initializer list significantly, so we won’t recommend their use as a best practice at this time.

Also, because there’s no enforcement that designated initializers are being used consistently everywhere an aggregate is initialized, it’s a good idea to avoid adding new members to the middle of an existing aggregate definition, to avoid the risk of initializer shifting.

Best practice

When adding a new member to an aggregate, it’s safest to add it to the bottom of the definition list so the initializers for other members don’t shift.

Assignment with an initializer list

As shown in the prior lesson, we can assign values to members of structs individually:

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 32, 60000.0 };

    joe.age  = 33;      // Joe had a birthday
    joe.wage = 66000.0; // and got a raise

    return 0;
}
```

This is fine for single members, but not great when we want to update many members. Similar to initializing a struct with an initializer list, you can also assign values to structs using an initializer list (which does memberwise assignment):

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 32, 60000.0 };
    joe = { joe.id, 33, 66000.0 }; // Joe had a birthday and got a raise

    return 0;
}
```

Note that because we didn't want to change `joe.id`, we needed to provide the current value for `joe.id` in our list as a placeholder, so that memberwise assignment could assign `joe.id` to `joe.id`. This is a bit ugly.

Assignment with designated initializers C++20

Designated initializers can also be used in a list assignment:

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 32, 60000.0 };
    joe = { .id = joe.id, .age = 33, .wage = 66000.0 }; // Joe had a birthday and got
a raise

    return 0;
}
```

Any members that aren't designated in such an assignment will be assigned the value that would be used for value initialization. If we hadn't have specified a designated initializer for `joe.id`, `joe.id` would have been assigned the value 0.

Initializing a struct with another struct of the same type

A struct may also be initialized using another struct of the same type:

```
#include <iostream>

struct Foo
{
    int a{};
    int b{};
    int c{};
};

int main()
{
    Foo foo { 1, 2, 3 };

    Foo x = foo; // copy initialization
    Foo y(foo);  // direct initialization
    Foo z {foo}; // list initialization

    std::cout << x.a << ' ' << y.b << ' ' << z.c << '\n';

    return 0;
}
```

The above prints:

```
1 2 3
```

Note that this uses the standard forms of initialization that we're familiar with (copy, direct, or list initialization) rather than aggregate initialization.

This is most commonly seen when initializing a struct with the return value of a function that returns a struct of the same type. We cover this in more detail in lesson 13.10 -- Passing and returning structs.