Use Cases

Concise initialization of simple `struct`s

Default member initializers provide a concise and effective way of initializing all the members of a simple **`struct`**. Consider, for instance, a **`struct`** used to configure a thread pool:

```
struct ThreadPoolConfiguration
{
    int  d_numThreads         = 8;     // number of worker threads
    bool d_enableWorkStealing = true;  // enable work stealing
    int  d_taskSize           = 64;    // buffer size for an enqueued task
};
```

Compared to the use of a constructor, the above definition of `ThreadPoolConfiguration` provides sensible default values with minimal **boilerplate code**.[1]

Ensuring initialization of a non`static` data member

~~N~~on**`static`** data members that do not have a default member initializer ~~or~~ appear in ~~any~~ constructor member initializer list are **default initialized**. For **user-defined types**, default initialization is equivalent to the default constructor ~~being invoked~~. For built-in types, default initialization results in an indeterminate value.

~~As an example, c~~onsider a **`struct`** tracking the number of times a user accesses a website:

```
#include <string>  // std::string

struct UsageTracker
{
    std::string d_token;
    std::string d_websiteURL;
    int         d_clicks;
};
```

The programmer intended `UsageTracker` to be used as a simple aggregate. Forgetting to explicitly initialize `d_clicks` ~~can result~~ in a defect:

[1]In C++20, designated initializers can be used to tweak one or more default settings in a configuration **`struct`** like `ThreadPoolConfiguration` in a clear and concise manner:

```
void testDesignatedInitializer()
{
    ThreadPoolConfiguration tpc = {.d_taskSize = 128};
    assert(tpc.d_numThreads == 8);
    assert(tpc.d_enableWorkStealing);
    assert(tpc.d_taskSize == 128);
}
```

```
#include <map>     // std::map
#include <vector>  // std::vector

std::map<std::string, std::vector<UsageTracker>> usageTrackers;
// ...

void onVisitWebsite(const std::string& username, const std::string& token)
{
    UsageTracker ut = {token, "https://emcpps.com"};
    usageTrackers[username].push_back(ut);
        // Bug, ut.d_clicks has indeterminate value.
}
```

Consistent use of default member initializers for built-in types can avoid such defects:

```
#include <string>  // std::string

struct UsageTracker
{
    ~~std::string d_token;~~
    ~~std::string d_websiteURL;~~
    int         d_clicks = 0;  // OK, will not have an indeterminate value
};
```

Avoiding boilerplate repetition across multiple constructors

Certain member variables of a type might be used to track the state of the object during its lifetime, independently of the initial state of the object. In such cases, we might want all constructors to set such variables to the same value, irrespective of constructor arguments. Consider a state machine that controls execution of a background process:

```
class StateMachine
{
    enum State { e_INIT = 1, e_RUNNING, e_DONE, e_FAIL };
    State          d_state;
    MachineProgram d_program;  // instructions to execute

public:
    StateMachine()  // Create a machine to run the default program.
    : d_state(e_INIT)
    , d_program(getDefaultProgram())
    { }

    StateMachine(const MachineProgram& program)  // Run the specified program.
    : d_state(e_INIT)
    , d_program(program)
    { }
```