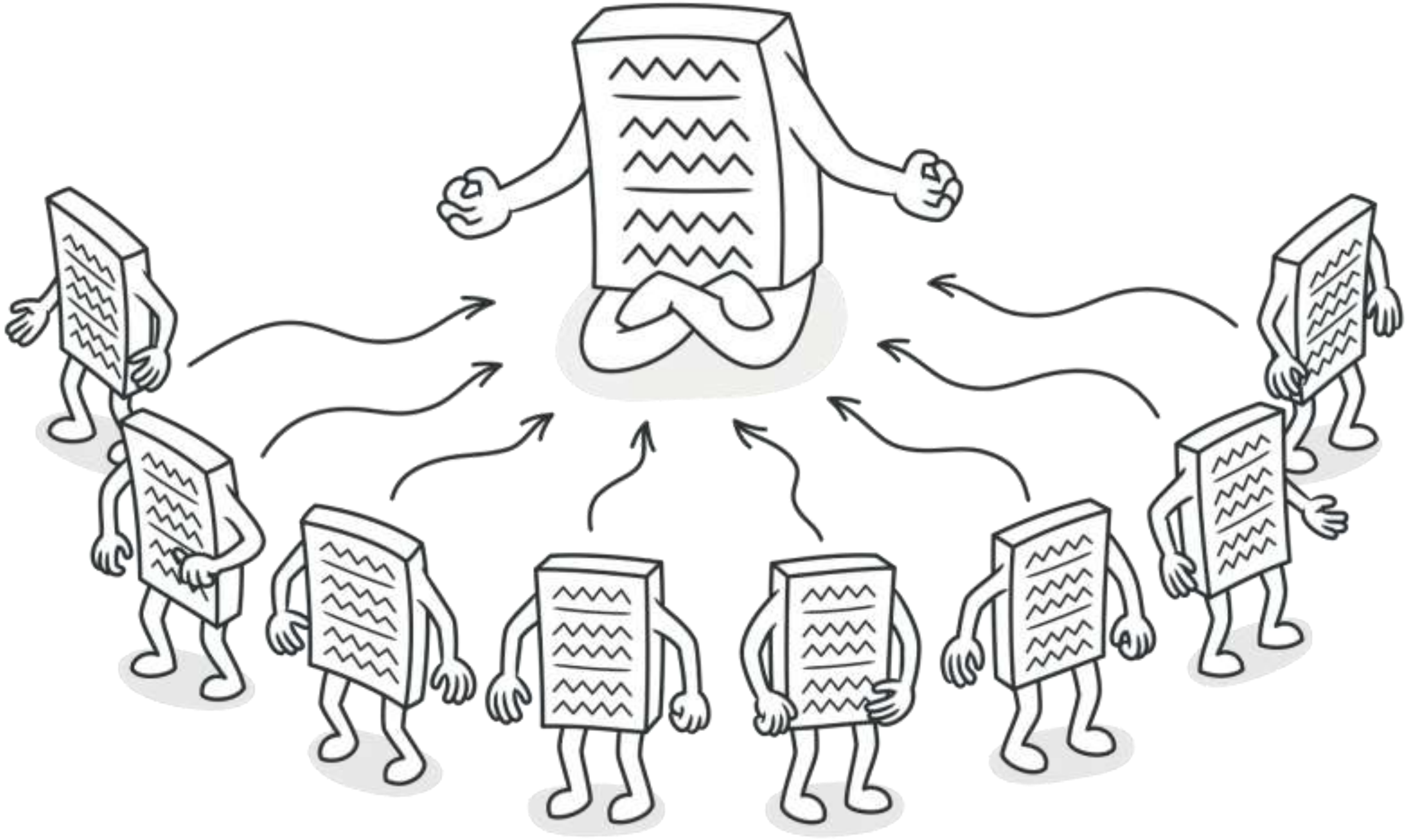
Singleton

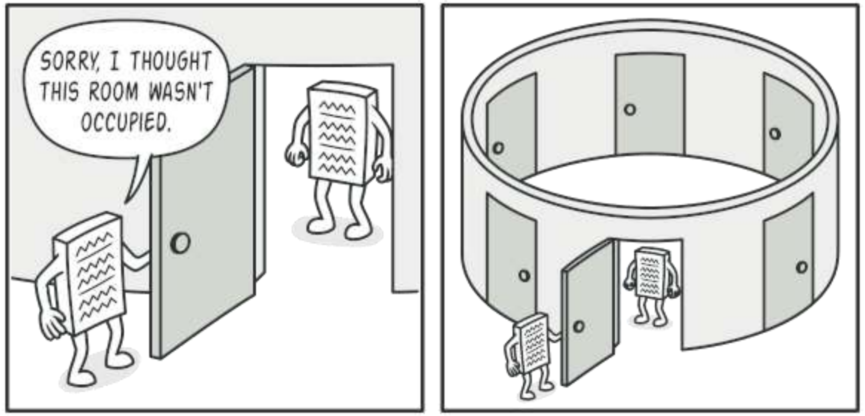


What is the singleton design pattern?

- Ensure a class only has <u>one</u> instance, and provide a <u>global point of access</u> to it." [GoF]

Primary reasons to use a singleton

- Multiple instances are a waste of resources or even undesirable
- Accessed across a large scope and multiple times
- Used in a similar fashion by all users
- Lazy initialization
- Examples
  - Logger
  - Filesystem
  - Design patterns:
    - (Abstract) Factory
    - Builder
    - Prototype
    - Facade
    - State
    - …



Building the perfect logger – Static class

```cpp
class Logger {
public:
    static void log(const std::string& msg) { /* log some message */}
private:
    Logger() = delete;
};
```

Suitable if we can cope with:

- No inheritance
- Not lazily initialized
- Cannot be passed around

Building the perfect logger – Basic singleton

```cpp
class Logger {
    inline static Logger * instance = nullptr;
public:
    static Logger& get_instance() {
        if(!instance) {
            instance = new Logger();
        }
        return *instance;
    }

    void log(const std::string& msg) { /* log some message */ }
private:
    Logger() { /* construction code */ }
    ~Logger() { /* destruction code */ }
};
```

```cpp
int main(int argc, char ** argv) {
    Logger::get_instance().log("Hi mom look at me presentating!");
}
```

Building the perfect logger – Copy problems

```
class Logger {
    inline static Logger * instance = nullptr;
public:
    static Logger& get_instance() {
        if(!instance) {
            instance = new Logger();
        }
        return *instance;
    }

    void log(const std::string& msg) { /* log some message */ }
private:
    Logger() { /* construction code */ }
    ~Logger() { /* destruction code */ }
};
```

```
int main(int argc, char ** argv) {
    Logger::get_instance().log("Hi mom look at me presentating!");
    Logger logger = Logger::get_instance();
    logger.log("Your T-shirt is just perfect, sweetie!");
}
```

☹ Creates new instance!

Building the perfect logger – Copy solution

```
class Logger {
    inline static Logger * instance = nullptr;
public:
    static Logger& get_instance() {
        if(!instance) {
            instance = new Logger();
        }
        return *instance;
    }

    void log(const std::string& msg) { /* log some message */ }
private:
    Logger() { /* construction code */ }
    ~Logger() { /* destruction code */ }

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;
    /* Rule of four – move constructor…*/
};
```

Explicit deletion means compiler does not implicitly generate them

```
int main(int argc, char ** argv) {
    Logger::get_instance().log("Hi mom look at me presentating!");
    Logger logger = Logger::get_instance();
    logger.log("Your T-shirt is just perfect, sweetie!");
}
```

Compiler to the rescue:
error: use of deleted function (or similar) ‘Logger::Logger(const Logger&)’
Logger logger = Logger::get_instance();

Building the perfect logger – Usage after copy solution

```
class Logger {
    inline static Logger * instance = nullptr;
public:
    static Logger& get_instance() {
        if(!instance) {
            instance = new Logger();
        }
        return *instance;
    }

    void log(const std::string& msg) { /* log some message */ }
private:
    Logger() { /* construction code */ }
    ~Logger() { /* destruction code */ }

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;
    /* Rule of four – move constructor…*/
};
```

```
int main(int argc, char ** argv) {
    Logger::get_instance().log("Hi mom look at me presentating!");
    Logger& logger = Logger::get_instance();
    logger.log("Your T-shirt is just perfect, sweetie!");
}
```

Logger& works
(auto doesn't, but
auto& does)

Building the perfect logger – Race

```
class Logger {
    inline static Logger * instance = nullptr;
public:
    static Logger& get_instance() {
        if(!instance) {
            instance = new Logger();
        }
        return *instance;
    }

    void log(const std::string& msg) { /* log some message */ }
private:
    Logger() { /* construction code */ }
    ~Logger() { /* destruction code */ }

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;
    /* Rule of four - move constructor…*/
};
```

☹ Race condition

```
int main(int argc, char ** argv) {
    Logger::get_instance().log("Hi mom look at me presentating!");
    Logger& logger = Logger::get_instance();
    logger.log("Your T-shirt is just perfect, sweetie!");
}
```

Building the perfect logger – Thread safe but slow

```
class Logger {
    inline static Logger * instance = nullptr;
    inline static std::mutex lock;
public:
    static Logger& get_instance() {
        std::lock_guard<std::mutex> guard(lock);
        if(!instance) {
            instance = new Logger();
        }
        return *instance;
    }
/* … */
};
```

Building the perfect logger – Seemingly thread safe and fast

```
class Logger {
    inline static Logger * instance = nullptr;
    inline static std::mutex lock;
public:
    static Logger& get_instance() {
        if(!instance) {
            std::lock_guard<std::mutex> guard(lock);
            if(!instance) {
                instance = new Logger();
            }
        }
        return *instance;
    }
/* … */
};
```

Building the perfect logger – Thread safe and fast

```
class Logger {
    inline static std::atomic<Logger*> instance = nullptr;
    inline static std::mutex lock;
public:
    static Logger& get_instance() {
        if(!instance) {
            std::lock_guard<std::mutex> guard(lock);
            if(!instance) {
                instance = new Logger();
            }
        }
        return *instance;
    }
/* … */
};
```

Building the perfect logger – Thread safety with once_flag

```
class Logger {
    inline static std::atomic<Logger*> instance = nullptr;
    inline static std::once_flag flag;
public:
    static Logger& get_instance() {
        std::call_once(flag, [&]() { instance = new Logger(); });
        return *instance;
    }
/* … */
};
```

Thread safety in C# (and a teaser for C++ later)

```
class Logger {
    public static Logger Instance{ get; };
    public void Log() { /* log some message */ }
    /* ... */
};
/* OR */
class Logger {
    static Lazy<Logger> instance;
    public static Logger Instance => instance.Value;
    public void Log() { /* log some message */ }
};

Logger.Instance.Log("My top level message");
```

Inheritance

- Does/can the base class need to explicitly know about all of the inheritors?
- Do we want only a single instance overall or is an instance of every inheritor suitable?
- Should the base class be abstract?
- Can we replace the instance with a different inheritor instance?

Building the perfect logger – Inheritance with predefined types

```
class Logger {
/* … */
public:
    enum class type {JSON, DEFAULT };

    static Logger& get_instance() {
        if(!instance) {
            switch(Config::LoggerType) {
                case type::JSON:
                    instance = new JSONLogger();
                    break;
                case type::DEFAULT:
                default:
                    instance = new Logger();
            }
        }

        return *instance;
    }
    virtual void log(const std::string& msg) { /* Default log implementation */ }
protected:
    virtual ~Logger() { /* destruction code */ }
/* … */
};
```

```
class JSONLogger : public Logger
{
public:
    virtual void log(const std::string& msg) { /* Default log implementation */ }
private:
    JSONLogger() : Logger() { /* JSON logger specific construction */}
    ~JSONLogger() { /* JSON logger specific destruction */}
    friend class Logger;
};
```

Allows calling the private constructor

Building the perfect logger – Inheritance with a registry

```cpp
class Logger {
/* … */
    inline static std::unordered_map<std::string, Logger*> registry;
public:
    static Logger& get_instance() {
        if(!instance) {
            instance = registry[Config::LoggerType()];
        }
        return *instance;
    }
    virtual void log(const std::string& msg) { /* Default log implementation */ }
protected:
    static void register_child(const std::string& name, Logger * inst) { registry[name] = inst;}
    Logger() { /* construction code */ }
    virtual ~Logger() { /* destruction code */ }
/* … */
};
```

```cpp
class JSONLogger : public Logger
{
public:
    virtual void log(const std::string& msg) { /* Default log implementation */ }
private:
    JSONLogger() : Logger() { /* JSON logger specific construction */}
    ~JSONLogger() { /* JSON logger specific destruction */}
    friend class Logger;
};
```

Building the perfect logger – Inheritance with templating

```
class Logger {
    inline static Logger* instance = nullptr;
public:

    template<typename T = Logger>
    static Logger& get_instance() {
        if (!instance) {
            instance = new T();
        }
        return *instance;
    }

    virtual void log(const std::string& msg) { /* Default log implementation */ }
protected:
    Logger() { /* construction code */ }
    virtual ~Logger() { /* destruction code */ }
/* … */

};
```

```
class JSONLogger : public Logger
{
public:
    virtual void log(const std::string& msg) { /* Default log implementation */ }
private:
    JSONLogger() : Logger() { /* JSON logger specific construction */}
    ~JSONLogger() { /* JSON logger specific destruction */}
    friend class Logger;
};
```

Friend class in C# (and why this lecture is C++ specific)

- No direct equivalent
- Possible equivalents with concessions:
  - Internal keyword
  - … with an attribute
  - Nested classes

```
[assembly: InternalsVisibleTo("OtherAssembly")]

partial class Outer
{
}

partial class Outer
{
    class Inner
    {
       // This class can access Outer's private members
    }
}
```

  - Interface coupling

…

Lifespan and destruction

- Who calls the destructor:
  - It can be left up to the language behavior
  - Or it can be done manually
- In what order are the objects destroyed
- How much of a problem are short time memory leaks

Ostrich singleton approach

- We solve the problem by not solving it
- Resources are freed when process terminates
- Resource leak is very short timewise

```
class Ostrich{
private:
        Ostrich() {};
        inline static OstrichData* instance = nullptr;
public:
        static Ostrich& getInstance();
        void doSomething();
};
```

Meyersen‘s singleton

- We can use a local static variable
- Instantiated at first parent method call
- Thread safe

```
class Logger {
public:
    static Logger& get_instance() {
        static Logger instance;
        return instance;
    }

    void log(const std::string& msg) { /* log some message */}
private:
    Logger() { /* construction code */ }
    ~Logger() { /* destruction code */}

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;
};
```

Static local variables

```
static Logger& get_instance() {
    static Logger instance;
    return instance;
}
```

Roughly translates to

```
Logger&  get_instance() {
   extern void __constructDatabase(void* memory);
   extern void __destroyDatabase();

   static bool __initialized = false;
   static char __buffer[sizeof(Logger)];

   if (!__initialized) {
     __constructDatabase(__buffer);
     atexit(__destroyDatabase);
     __initialized = true;
   }
   return *reinterpret_cast<Logger*>(__buffer);
}
```

Initialization only done after invocation

Called on exit for destruction purposes in LIFO

LIFO limitations

If objects are destroyed in LIFO order with respect to creation, the following can take place:

1. Keyboard init
2. Display init with error
3. Logger init and error logged
4. Program end
5. Logger destruction
6. Display destruction with error
7. **Logger is already destroyed**

Checking the validity of the instance

We can simply add a flag paired with the instance holder, which signifies if the instance already went through destruction

```
class Logger {
    inline static Logger* instance = nullptr;
    inline static bool destroyed = false;
public:
    static Logger& get_instance() {
        if (!instance) {
            if (destroyed) {
                on_destroyed();
             }
             else {
                 create();
             }
         }
         return *instance;
    }
private:
    static void create() {
        static Logger staticInstance;
        instance = &staticInstance;
    }
    static void on_destroyed() { /* ? */ }
    ~Logger() {
        destroyed = true;
        instance = nullptr;
    }
/* ... */
};
```

What to do on `on_destroyed()`?

- Throw an exception
  - Helps by not helping
  - Potentially throws an exception in a destructor
- Recreate the singleton instance – Phoenix singleton

```
static void killPhoenix() {
    instance->~Logger();
}
static void on_destroyed() {
    create();
    new(instance) Logger;
    atexit(killPhoenix);
    destroyed = false;
}
```

The new instance is created in the same memory as the old one

We need to register the instance for destruction again

Evade `on_destroyed()` completely

- A possible solution is to add longevity to objects with dependant destruction
- Priority queue over longevity

jak evidovat frontu - tech detaily, mechanismus type erasure

```
class NonSingleton { /**/ };
class KeyboardSingleton { /**/ };
class LogSingleton { /**/ };

NonSingleton* non_singleton(new NonSingleton);

template<typename T>
void setLongevity(T* object, int longevity);
template<typename T>
void setLongevity(T& object, int longevity);

int main() {
    setLongevity(non_singleton, 5);
    setLongevity(KeyboardSingleton::instance(), 5);
    setLongevity(LogSingleton::instance(), 6);
}
```

Policies

- We can create a general singleton through templates and so called policies
- Possible policies:
  - Creation – new, malloc, inheritance
  - Lifetime – ostrich, phoenix, longevity
  - Thread safety – single, multithreaded, other

Policies

```cpp
template
<
    class T,
    template <class> class CreationPolicy = CreateUsingNew,
    template <class> class LifetimePolicy = DefaultLifetime,
    template <class> class ThreadingModel = SingleThreaded
>
class SingletonHolder {
    using InstanceType = ThreadingModel<T>::VolatileType;
    static InstanceType* instance;
    static bool destroyed;
public:
    static T & get_instance();
private:
    static void destroy();
    SingletonHolder();
/* ... */
};
```

```cpp
class Logger {
    public:
        void log(const std::string& msg) { /* log some message */ }
};

    using defaultLogger = SingletonHolder<Logger>;
    using threadSafePhoenixLogger = SingletonHolder<
        Logger,
        CreateUsingNew,
        PhoenixLifetime,
        MultiThreaded>;

    void main() {
        defaultLogger::get_instance().log("basic log");
        defaultLogger::get_instance().log("destruction proof and
                                          thread safe log");
    }
```

Pros and cons

☺ Ensures the single instance requirement

☺ Lazy initialization

☹ Violates the single responsibility principle

☹ Increases coupling

☹ Shown problems with multithreading and destruction

☹ Private constructor and other singleton properties make testing difficult

☺ Very useful in cirmustances that exactly fit