Decoupling Software

Techniques and Benefits

David Sackstein

Agenda

- Coupling: The butterfly effect
- Compiler, Linker and Loader
- Libraries: static, shared objects, dlls
- Techniques to reduce coupling
  - Single Responsibility
  - Programming to an API (limit visibility)
  - Prefer Composition over Inheritance
  - The pimpl idiom
  - Using interfaces
  - Dynamic linking
  - Dynamic loading and plugins

Coupling: The butterfly effect

- When software components are coupled:
  - Changes to one may require changes in another.
  - Changes to one may require building the entire project.
  - Bugs are difficult to isolate because the bug might have been caused by the original change or by any of the resulting changes.
  - Just like the compiler and linker need to build the whole project to make sure it is consistent, so also, programmers may need to download and read the whole project to understand one component.

Compiler

- The compiler converts a C/C++ file (also called a translation unit) to an object file.
- Code in the object file may call functions and refer to external symbols that are not defined in the same translation unit.
- The compiler adds metadata to the object file to list the symbols defined in the translation unit and to list the symbols referenced.

Linker

- The linker converts source files to a set of object files (and libraries) and creates an executable image.
  - ELF on Linux
  - PE on Windows
- An executable is one of two
  - An exe (has a main entry point)
  - A dynamically linked library (no main)
- On Linux an exe can also be used as a library
- The linker resolves all symbols and updates call addresses.
- The linker fails if there are multiple symbols with the same name or if there are missing symbols.

Loader

- The loader is part of the operating system
  - Loads the executable file (ELF, PE) into memory.
  - Allocates memory for program segments (code, data, heap, stack).
  - Resolves addresses for dynamic libraries if used.
  - Sets up the process environment (arguments, environment variables, file descriptors).
  - Transfers control to the program’s entry point (main or initialization code).

Libraries

- There are two types of libraries for C/C++
  - Static libraries
    - On Linux often called archives (extension .a)
    - On Windows often called libraries (extension .lib)
  - Dynamic libraries
    - On Linux called Shared Objects (extension .so)
    - On Windows called Dynamically Linked Library (or DLL with extension .dll)

Static Libraries

- An archive of compiled object files created by the linker.
- Does not define a main (entry point)
- The linker does not resolve symbols for the library (some symbols may remain unresolved)
- When linked to an executable, the linker resolves all the symbols of all object files and libraries together.

Static Libraries: Pros and Cons

- Pros
  - Code organization
  - Reduced build time
  - Hiding implementations from non-IDA-experts
- Cons
  - Dependency requirements leak out of the library.
  - All symbols in a static library are exported (no encapsulation)
  - Each executable image that uses the library contains its own copy of the library (no savings in disk or memory footprint)

Shared Objects

- An ELF file (like an executable) created by the linker from object files but with no main (entry point)
- Unlike static libraries, the linker does resolve symbols for the library.
- All symbols used in the library must be provided to the linker of the library (in an object or library file)
- On Linux all symbols by default (unlike Windows) but you can change the default and expose selectively
- Linkage of a shared object is partially postponed to the load time. Hence: Dynamic Loading
- Only the name of the shared object and the method names are stored.

Dynamically Linked Libraries

- When the linker links an executable, it accepts object files, static libraries and dynamic libraries.
- References to symbols found in dynamic libraries are inserted into a table that is read by the loader together with the name of the library.
- When the executable is loaded, the loader reads the table, locates the DLL, loads it and finally links all references.
- This is dynamic linking because it occurs at load time.
- Many executables can use the same DLL (it is not included in the executable image).

Shared Objects: Pros

- All the pros of static libraries
- Encapsulation: You decide which symbols are exported
- Reduced disk imprint: The shared object image on disk can be shared by multiple applications
- Reduce memory imprint: When using the –fPIC option to build the shared object, the code of the so is also shared in memory
- You can still offer different versions of the library using different SONAMEs

Shared Objects: Cons

- Slight increase in load time
- Slight increase in first call to each method at run time.
- Managing the load library path in a deployment is a little tricky

Techniques to reduce coupling

- Assume (the worst :)
  - A needs to use B (B is a dependency of A)
  - Other classes in the project use A
  - A inherits B (bad inheritance).
  - Both are defined in the same files, AB.h and AB.cpp
- Whenever anything in B changes:
  - We must compile AB.cpp again
  - There is a risk that we might unintentionally change A too because it is defined in the same files.
  - If AB.h changes, then we have a butterfly effect.
  - We may have to recompile and build the whole project.

Apply the SRP

- Each class should be in its own header file.
- Each class should be in its own source file.
- Now it is unlikely that we will unintentionally change A when we change B.
- But:
  - If the API of B changes we must change A and recompile A.
  - Much worse: As A inherits B then all public members of B are visible to those who use A.
  - We have the butterfly effect again. Recompile the project.

Programming to an API (visibility)

- Separate the public and private parts of B
- Keep all implementation details of B private.
- All data fields of B (and A) should be private.
- If you still insist on inheriting B (you will soon change your mind), make the inheritance private.
- Now, if only the implementation of B changes, A's source code should not need to change. This is a good thing because public APIs are relatively stable.
- However:
  - If included headers changed, we may need to change the header and the source of A (at least to add missing headers).
  - Note that sometimes a change in the inclusions of B does not change the source code of B but changes its size.
  - As a result, we still need to recompile A and the whole project (why?).

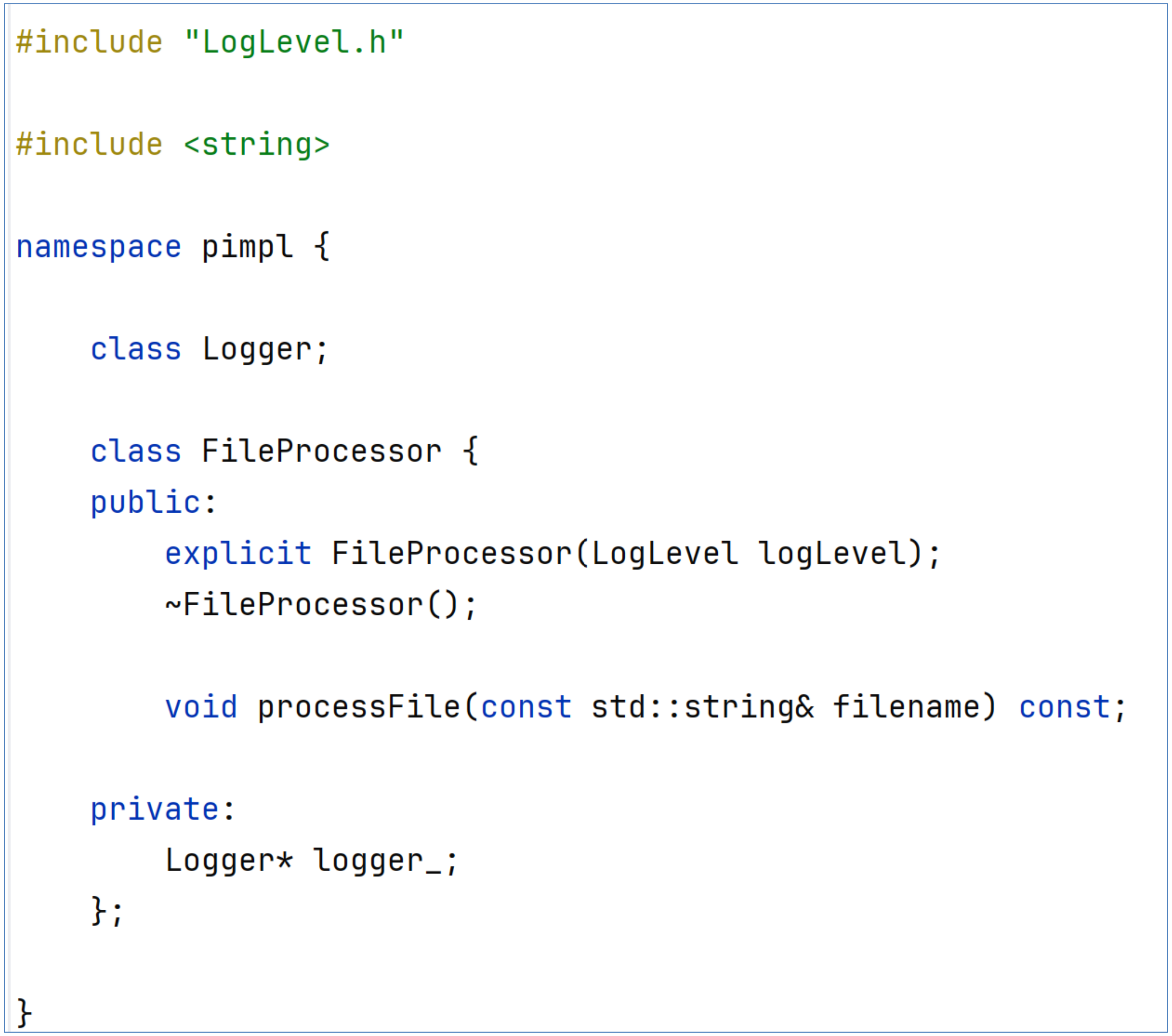
Prefer Composition over Inheritance

- Make the base class a pointer* to B
- Create the pointer in the constructor of A and delete it in the destructor of B
- The constructor of A should be in A.cpp not in A.h (why?)
- A's header can now forward declare B and omit inclusion of B.h in A.h. A will include B.h in A.cpp
- This is the pimpl idiom

The pimpl idiom

- Now:
  - Changes in B's implementation do not change the size of A and do not propagate to the rest of the project.
  - We have succeeded in isolating changes in B from the rest of the project
  - The pimpl idiom works well in C too
- But
  - A.cpp must still includes B.h
  - So, the implementation A still depends on the implementation of B (why?)
  - If included headers of B change, we may need to change A.cpp (at least to add missing headers)

Using pimpl there is no need to include Logger.h



```cpp
#include "LogLevel.h"

#include <string>

namespace pimpl {

    class Logger;

    class FileProcessor {
    public:
        explicit FileProcessor(LogLevel logLevel);
        ~FileProcessor();

        void processFile(const std::string& filename) const;

    private:
        Logger* logger_;
    };

}
```

Programming by Interface

- But in C++ we can do better: interfaces
- An interface is the public API of a class with all implementation details completely stripped.
- Instead of a forward declaration, include the header that defines the interface (IB.h)
- Accept the interface in the constructor of A and store it as a member.
- Now:
  - Changes in B.h have no effect on A at all, and therefore they have no effect on the rest of the project through A.
- But:
  - Changes in B do require relinking of B

What is an interface?

- An interface in C++ should:
  - Have only public pure virtual methods.
  - A virtual destructor (public or protected)
  - Have NO data fields, not even private ones.
  - Have NO inline implementations
- You can emulate interfaces in C by defining a struct with fields that are pointers to functions.

The ILogger Interface

```cpp
enum class LogLevel {
    DEBUG,
    INFO,
    ERROR
};

class ILogger {
public:
    virtual ~ILogger() = default;

    virtual void log(LogLevel, const std::string& msg) = 0;
};
```

Decoupled linking with Shared Objects

- How can we avoid relinking of the project when one of its classes is recompiled?
  - Move B out of the project into a shared library (B.so)
  - Moving B into a static library won’t help.
  - Expose (export) one factory method only from the library.
  - Do not expose other symbols (encapsulation)
  - The factory method creates B and returns its interface.
  - The main application creates a pointer to B and provides it to A’s constructor.
  - Changes to B require relinking of B.so but not of A’s project.

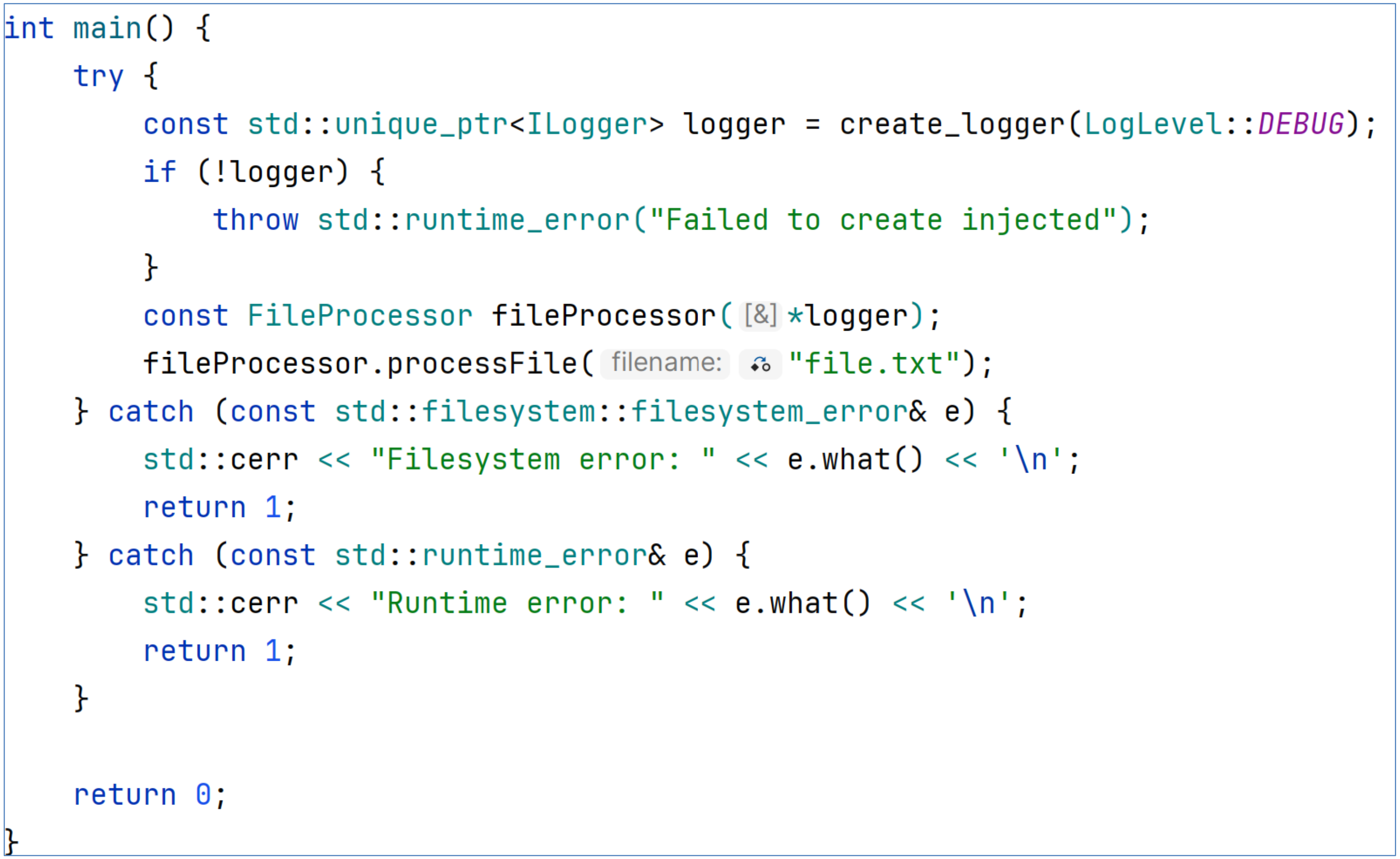
Decoupling at run time

- We can now change the implementation of B at deploy time. Just replace B.so in the load library path of the project.
- This enables partial updates of a deployed product.
- But A still needs to link to the name of the B.so library and to its exposed symbols (for the factory function).
  - These must be known when A's project is built and must not be changed
  - The library B.so with the linked name, must be present when the project runs.

Dynamic loading and plugins

- Dynamic loading using dlopen and dlsym
  - A’s project does not link to B.so at all
  - Instead, to create an IB the main request the library using dlopen by its name, and the factory method within the library by its name.
  - If they are not present, the main can decide whether it can continue without them.
  - Moreover, the main can scan a path for libraries that contain the required method – and load it.
  - This provides a plugin capability

Fully decoupled



```cpp
int main() {
    try {
        const std::unique_ptr<ILogger> logger = create_logger(LogLevel::DEBUG);
        if (!logger) {
            throw std::runtime_error("Failed to create injected");
        }
        const FileProcessor fileProcessor( [&] *logger);
        fileProcessor.processFile( filename: "file.txt");
    } catch (const std::filesystem::filesystem_error& e) {
        std::cerr << "Filesystem error: " << e.what() << '\n';
        return 1;
    } catch (const std::runtime_error& e) {
        std::cerr << "Runtime error: " << e.what() << '\n';
        return 1;
    }

    return 0;
}
```

Dynamic Linking vs Loading

- Dynamic linking requires that the DLL or shared object be present when the program loads. On the other hand, dynamic loading is done programmatically and can avoid loading the library if it is not present (useful for plugins and dependency injection).
- However, dynamic linking requires specialized code to load and link symbols so it is a little more work.
- Use of dynamic loading is rare.

Header Organization

- A header should define only one thing (SRP)
- A header should not include headers it does not need.
- A header should include all the headers it does need; this relieves users from the need to include them.
- Do not use forward declarations for types needed in the public API (for the same reason)
- Include specific headers before general headers. Why?
- Implementation headers and API headers are different.
  - Place them in different directories.
- Preferably do not inline implementations in headers

Key Takeaways

Decoupled code	Is easier to test, to maintain and to extend
pimpl Idiom	Hide implementation behind a pointer
Interface	Hide implementation behind a contract
Composition	Use dependency injection of interface ptrs
Shared Objects	Dynamic Linking isolates libraries
Headers	Headers represent dependencies.
	Manage your includes to prevent butterfly effects.