HIDING IMPLEMENTATION DETAILS

John Franklin Rickard | john@moduleworks.com

Good Code

Good Code

- Correct

Good Code

- Correct
- Performant

Good Code

- Correct
- Performant
- Maintainable

Good Code

- Correct
- Performant
- Maintainable
- Testable

Good Code

- ~~Correct~~
- Performant
- Maintainable
- Testable

Good Code

- ~~Correct~~
- ~~Performant~~
- Maintainable
- Testable

Good Code

- ~~Correct~~
- ~~Performant~~
- ~~Maintainable~~
- Testable

Good Code

- ~~Correct~~
- ~~Performant~~
- ~~Maintainable~~
- ~~Testable~~

~~Good Code~~

- ~~Correct~~
- ~~Performant~~
- ~~Maintainable~~
- ~~Testable~~

Managing Complexity

Managing Complexity

```
// TODO
```

Managing Complexity

```
// TODO
```

```
// FIXME
```

Managing Complexity

```
// TODO
```

```
// FIXME
```

```
// Do not touch
```

Managing Complexity

```
// TODO
```

```
// FIXME
```

```
// Do not touch
```

```
// !!! PLEASE DO NOT TOUCH !!!
```

Reasons To Hide Implementations

Reasons To Hide Implementations

- Complexity

Reasons To Hide Implementations

- Complexity
- Verbosity

Reasons To Hide Implementations

- Complexity
- Verbosity
- Dependencies

Reasons To Hide Implementations

- Complexity
- Verbosity
- Dependencies
- Secrets

Reasons To Hide Implementations

- Complexity
- Verbosity
- Dependencies
- Secrets
- Names

Functions

Functions

```
1 int dow(int m, int d, int y) {
2     y -= m<3;
3     return (y+y/4-y/100+y/400+"-bed=pen+mad."[m]+d) % 7;
4 }
```

Functions

```
1 int dayofweek(int m, int d, int y) {
2     y -= m<3;
3     return (y+y/4-y/100+y/400+"-bed=pen+mad."[m]+d) % 7;
4 }
```

Functions

```
1 // returns 0=Sunday, 1=Monday, etc.
2 int dayofweek(int m, int d, int y) {
3     y -= m<3;
4     return (y+y/4-y/100+y/400+"-bed=pen+mad."[m]+d) % 7;
5 }
```

Functions

```
1 // returns 0=Sunday, 1=Monday, etc.
2 int dayofweek(int m, int d, int y) {
3     y -= m<3;
4     return (y+y/4-y/100+y/400+"-bed=pen+mad."[m]+d) % 7;
5 }
```

- Functions should be your first line of defence

Functions

```
1 // returns 0=Sunday, 1=Monday, etc.
2 int dayofweek(int m, int d, int y) {
3     y -= m<3;
4     return (y+y/4-y/100+y/400+"-bed=pen+mad."[m]+d) % 7;
5 }
```

- Functions should be your first line of defence
- Naming something gives it power (even if its the wrong name!)

Split Declarations

```
1 // file: dayofweek.hpp
2
3 // returns 0=Sunday, 1=Monday, etc.
4 int dayofweek(int m, int d, int y);
```

Split Declarations

```
1 // file: ExpensiveNumber.hpp
2
3 inline float ExpensiveNumber() {
4     // a few lines of code ...
5     return result;
6 }
```

Split Declarations

```
1 // file: ExpensiveNumber.hpp
2 #include "boost/geometry.h"
3
4 inline float ExpensiveNumber() {
5     // a few lines of code ...
6     return result;
7 }
```

Split Declarations

```
1 // file: ExpensiveNumber.hpp
2
3 float ExpensiveNumber();
```

Abstractions With State

Abstractions With State

```
1 // file: Generator.hpp
2 class Generator {
3 public:
4     int RandomNumber() {
5         return m_value++;
6     }
7 private:
8     int m_value = 0;
9 };
```

Abstractions With State

```
1 // file: Generator.hpp
2 class Generator {
3 public:
4     int RandomNumber();
5 private:
6     int m_value = 0;
7 };
```

Abstractions With State

```
1 // file: Generator.hpp
2 #include "OtherGenerator.hpp"
3
4 class Generator {
5 public:
6     int RandomNumber();
7 private:
8     OtherGenerator m_generator;
9 };
```

Abstractions With State

```
1 // file: Generator.hpp
2 #include "OtherGenerator.hpp"
3
4 class Generator {
5 public:
6     int RandomNumber();
7 private:
8     OtherGenerator m_generator;
9 };
```

Unwanted dependency for customers!

Interfaces

Interfaces

```
1 // file: IGenerator.hpp
2 class IGenerator {
3 public:
4     virtual int RandomNumber() = 0;
5     virtual ~IGenerator() = default;
6 };
```

Interfaces

```
1 // file: IGenerator.hpp
2 class IGenerator {
3 public:
4     virtual int RandomNumber() = 0;
5     virtual ~IGenerator() = default;
6 };
```

```
// Hides the actual type from the user
std::unique_ptr<IGenerator> Create();
```

Interfaces

```
1  // file: Generator.hpp
2  #include "IGenerator.hpp"
3  #include "OtherGenerator.hpp"
4
5  class Generator : public IGenerator {
6  public:
7      int RandomNumber() override;
8  private:
9      OtherGenerator m_generator;
10 };
```

Pointers & References

```
1 struct FooImpl;
2
3 struct Foo {
4     FooImpl m_impl;
5 };
```

Pointers & References

```
1 struct FooImpl;
2
3 template<int i>
4 struct Foo {
5     FooImpl m_impl;
6 };
```

Pointers & References

```
1 struct FooImpl;
2
3 template<int i>
4 struct Foo {
5     FooImpl m_impl;
6 };
```

From cppreference:

> *A class template by itself is not a type, or an object, or any other entity. […] In order for any code to appear, a template must be instantiated*

Pointers & References

```
1 struct FooImpl;
2
3 struct Foo {
4     FooImpl& m_impl1;
5     FooImpl* m_impl2;
6 };
```

Opaque Pointers

Opaque Pointers

```
1 // file: FooImpl.hpp
2 struct FooImpl;
3 void InitFoo(FooImpl*& ptr)
4 void DoFooThings(FooImpl* ptr);
5 void DestroyFoo(FooImpl*& ptr);
```

Opaque Pointers

```
1 // file: FooImpl.hpp
2 struct FooImpl;
3 void InitFoo(FooImpl*& ptr)
4 void DoFooThings(FooImpl* ptr);
5 void DestroyFoo(FooImpl*& ptr);
```

```
1 // file: RandomUserFile.cpp
2 inline void works() {
3     FooImpl* ptr = nullptr;
4     InitFoo(ptr);
5     DoFooThings(ptr);
6     DestroyFoo(ptr);
7 }
```

Opaque Pointers

```
1 // file: FooImpl.hpp
2 struct FooImpl;
3 void InitFoo(FooImpl*& ptr)
4 void DoFooThings(FooImpl* ptr);
5 void DestroyFoo(FooImpl*& ptr);
```

```
1 // file: RandomUserFile.cpp
2 inline void works() {
3     FooImpl* ptr = nullptr;
4     InitFoo(ptr);
5     DoFooThings(ptr);
6     DestroyFoo(ptr);
7 }
```

Please don’t do this…

Pimpl Idiom

Pimpl Idiom

From cppreference:

> *“Pointer to implementation” […] removes implementation details of a class from its object representation by placing them in a separate class, accessed through an opaque pointer*

Pimpl Idiom

```
1 // file: Generator.hpp
2 class Generator {
3 public:
4     int RandomNumber() {
5         return m_value++;
6     }
7 private:
8     int m_value = 0;
9 };
```

Pimpl Idiom

```
1  // file: Generator.
2  struct GeneratorImpl;
3  class Generator {
4  public:
5      Generator();
6      int RandomNumber();
7      ~Generator();
8  private:
9      GeneratorImpl* m_pimpl;
10 };
```

Pimpl Idiom

```
1  // file: Generator.
2  struct GeneratorImpl;
3  class Generator {
4  public:
5      Generator();
6      int RandomNumber();
7      ~Generator();
8  private:
9      GeneratorImpl* m_pimpl;
10 };
```

Pimpl Idiom

```
1  // file: Generator.
2  struct GeneratorImpl;
3  class Generator {
4  public:
5      Generator();
6      int RandomNumber();
7      ~Generator();
8  private:
9      GeneratorImpl* m_pimpl;
10 };
```

Pimpl Idiom

```
1  // file: Generator.
2  struct GeneratorImpl;
3  class Generator {
4  public:
5      Generator();
6      int RandomNumber();
7      ~Generator();
8  private:
9      GeneratorImpl* m_pimpl;
10 };
```

Pimpl Idiom

```
1  // file: Generator.
2  struct GeneratorImpl;
3  class Generator {
4  public:
5      Generator();
6      int RandomNumber();
7      ~Generator();
8  private:
9      GeneratorImpl* m_pimpl;
10 };
```

Pimpl Idiom

```
1  // file: Generator.cpp
2  #include "Generator.hpp"
3  struct GeneratorImpl {
4      int m_value = 0;
5  };
6  Generator::Generator() : m_pimpl(new GeneratorImpl()) {
7  }
8  int Generator::RandomNumber() {
9      return m_pimpl->m_value++;
10 }
11 Generator::~Generator() {
12     delete m_pimpl;
13 }
```

Pimpl Idiom

```
1  // file: Generator.cpp
2  #include "Generator.hpp"
3  #include "OtherGenerator.hpp"
4  struct GeneratorImpl {
5      OtherGenerator m_generator;
6  };
7  Generator::Generator() : m_pimpl(new GeneratorImpl()) {
8  }
9  int Generator::RandomNumber() {
10     return m_pimpl->m_generator.RandomNumber();
11 }
12 Generator::~Generator() {
13     delete m_pimpl;
14 }
```

Pimpl Idiom

```
1  // file: Generator.hpp
2  struct GeneratorImpl;
3  class Generator {
4  public:
5      Generator();
6      Generator(const Generator&) = delete;
7      Generator(Generator&&) = delete;
8      Generator& operator=(const Generator&) = delete;
9      Generator& operator=(Generator&&) = delete;
10     int RandomNumber();
11     ~Generator();
12 private:
13     GeneratorImpl* m_pimpl;
14 };
```

Smart Pointers

```
1 // file: Generator.hpp
2 class Generator {
3 public:
4     Generator();
5     int RandomNumber();
6     ~Generator();
7 private:
8     std::unique_ptr<struct GeneratorImpl> m_pimpl;
9 };
```

Smart Pointers

```
1 // file: Generator.hpp
2 class Generator {
3 public:
4     Generator();
5     int RandomNumber();
6     ~Generator();
7 private:
8     std::unique_ptr<struct GeneratorImpl> m_pimpl;
9 };
```

```
1 // diff file: Generator.cpp
2 Generator() : m_pimpl(std::make_unique<GeneratorImpl>()){}
3 Generator::~Generator() = default;
```

Custom Deleter

```
1  // file: ImplDeleter.hpp
2  template <class T>
3  class ImplDeleter {
4  public:
5      ImplDeleter() noexcept : m_deleter(&DeleteImpl) {}
6      void operator()(T* ptr) const noexcept { m_deleter(ptr
7  private:
8      void (m_deleter)(T ptr);
9      static void DeleteImpl(T* ptr) noexcept {
10         static_assert(sizeof(T) > 0,
11             "[ImplDeleter]: T has to be complete here");
12         delete ptr;
13     }
14 };
```

Custom Deleter

```
1  // file: ImplDeleter.hpp
2  template <class T>
3  class ImplDeleter {
4  public:
5      ImplDeleter() noexcept : m_deleter(&DeleteImpl) {}
6      void operator()(T* ptr) const noexcept { m_deleter(pt
7  private:
8      void (m_deleter)(T ptr);
9      static void DeleteImpl(T* ptr) noexcept {
10         static_assert(sizeof(T) > 0,
11             "[ImplDeleter]: T has to be complete here");
12         delete ptr;
13     }
14 };
```

```
template<class T>
using PimplPtr = std:unique_ptr<T, ImplDeleter<T>>;
```

Custom Deleter

```
1 // file: Generator.hpp
2 class Generator {
3 public:
4     Generator();
5     int RandomNumber();
6 private:
7     PimplPtr<struct GeneratorImpl> m_pimpl;
8 };
```

Const Correctness

```
1 // file: Foo.hpp
2 class Foo {
3 public:
4     Foo();
5     void Get();
6     void Get() const;
7 private:
8     PimplPtr<struct FooImpl> m_pimpl;
9 };
```

Const Correctness

```
1  // file: Foo.cpp
2  #include "Foo.hpp"
3  class FooImpl {
4      void Get() {
5          std::cout << "Get" << std::endl;
6      }
7      void Get() const {
8          std::cout << "Const Get" << std::endl;
9      }
10 };
11 Foo() : m_pimpl(std::make_unique<FooImpl>()) {}
12 void Foo::Get() { return m_pimpl->Get(); }
13 void Foo::Get() const { return m_pimpl->Get(); }
```

Const Correctness

```
1 #include "Foo.hpp"
2 int main() {
3     auto f1 = Foo();
4     f1.Get();
5     const auto f2 = Foo();
6     f2.Get();
7 }
```

Const Correctness

```
1 #include "Foo.hpp"
2 int main() {
3     auto f1 = Foo();
4     f1.Get();
5     const auto f2 = Foo();
6     f2.Get();
7 }
```

Prints non const “Get” both times!

Const Correctness

```
1  #include <concepts>
2
3  struct Bar{ int* m_ptr; };
4
5  static_assert(requires (Bar b) {
6      {*b.m_ptr} -> std::same_as<int&>;
7  });
8  static_assert(requires (const Bar b) {
9      {*b.m_ptr} -> std::same_as<int&>;
10 });
```

Const Correctness

```
// file: Foo.hpp
class Foo {
    ...
    FooImpl& Impl();
    const FooImpl& Impl()const ;
};
```

Const Correctness

```
// file: Foo.hpp
class Foo {
    ...
    FooImpl& Impl();
    const FooImpl& Impl()const ;
};
```

```
// file: Foo.cpp
...
FooImpl& Foo::Impl() { return *m_pimpl; }
const FooImpl& Foo::Impl() const { return *m_pimpl; }
void Foo::Get() { return Impl().Get(); }
void Foo::Get() const { return Impl().Get(); }
```

Const Correctness

Modern C++ Solution

… soon ?!

```
template<class T>
using PimplPtr = std::experimental::propagate_const<std:uniqu
```

Pimpl Wrapper

```
1  template <class T>
2  class UniquePimpl {
3  public:
4      explicit UniquePimpl(T* implPtr) noexcept : m_ptr(impl
5      T& operator*() { return *m_ptr; }
6      const T& operator*() const { return *m_ptr; }
7      T* operator->()    { return m_ptr.get(); }
8      const T* operator->() const { return m_ptr.get(); }
9  protected:
10     using Ptr = std::unique_ptr<T, ImplDeleter<T>>;
11     explicit UniquePimpl(Ptr&& ptr) noexcept : m_ptr(std::
12     Ptr m_ptr;
13 };
```

Pimpl Wrapper Example

```
1 // file: Generator.hpp
2 class Generator {
3 public:
4     int RandomNumber() {
5         return m_value++;
6     }
7 private:
8     int m_value = 0;
9 };
```

Pimpl Wrapper Example

```
1 // file: Generator.hpp
2 #include "UniquePimpl.hpp"
3 class Generator {
4 public:
5     Generator();
6     int RandomNumber();
7 private:
8     UniquePimpl<struct GeneratorImpl> m_pimpl;
9 };
```

Pimpl Wrapper Example

```
1  // file: Generator.cpp
2  #include "Generator.hpp"
3  struct GeneratorImpl {
4      int m_value = 0;
5  };
6  Generator::Generator() : m_pimpl(new GeneratorImpl()) {
7  }
8  int Generator::RandomNumber() {
9      return m_pimpl->m_value++;
10 }
```

Pimpl Idiom

Pimpl Idiom

- Hides implementation fully

Pimpl Idiom

- Hides implementation fully
- Compilation firewall

Pimpl Idiom

- Hides implementation fully
- Compilation firewall
- ABI stable interface

Pimpl Idiom

- Hides implementation fully
- Compilation firewall
- ABI stable interface
- A lot of boilerplate code

Pimpl Idiom

- Hides implementation fully
- Compilation firewall
- ABI stable interface
- A lot of boilerplate code
- Slight performance regression

Cached Copy

```
1  // file: ImplCopier.hpp
2  template <class T>
3  class ImplCopier {
4  public:
5      ImplCopier() noexcept : m_copier(&CopyFunction) {}
6      [[nodiscard]] T* operator()(const T& impl) const { ret
7  private:
8      T* (*m_copier)(const T& impl);
9      static T* CopyFunction(const T& impl) {
10         static_assert(sizeof(T) > 0,
11             "[ImplCopier]: T has to be complete here");
12         return new T(impl);
13     }
14 };
```

Copyable Wrapper

```
1  // file: ValuePimpl.hpp
2  template <class T>
3  class ValuePimpl : public UniquePimpl<T> {
4  public:
5      explicit ValuePimpl(T* ptr) noexcept : UniquePimpl<T>
6      ValuePimpl(const ValuePimpl& other)
7          : UniquePimpl<T>(other.Copyptr()), m_copier(other.
8      ValuePimpl& operator=(const ValuePimpl& other) {
9          this->m_ptr = other.Copyptr();
10         m_copier = other.m_copier;
11         return *this;
12     }
13     ValuePimpl(ValuePimpl&& other) noexcept = default;
14     ValuePimpl& operator=(ValuePimpl&& other) noexcept = d
15     ...
```

Copyable Wrapper

```
1  // file: ValuePimpl.hpp
2  template <class T>
3  class ValuePimpl : public UniquePimpl<T> {
4      ...
5  private:
6      ImplCopier<T> m_copier;
7      using Ptr = typename UniquePimpl<T>::Ptr;
8      MW_NODISCARD Ptr Copyptr() const {
9          return Ptr(m_copier(*(*this)), this->m_ptr.get_del
10     }
11 };
```

Copyable Wrapper Example

```
1 // file: Generator.hpp
2 #include "ValuePimpl.hpp"
3 class Generator {
4 public:
5     Generator();
6     int RandomNumber();
7 private:
8     ValuePimpl<struct GeneratorImpl> m_pimpl;
9 };
```

Further Reading

- pimpl cppreference
- GOTW 100
- Pimpl, Rule Of Zero

THANK YOU FOR YOUR ATTENTION!