Object-Oriented Programming

PIMPL

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Encapsulation & Information Hiding Rules

Place data and the operations that perform on that data in the same class

- Don't expose data items
- Don't expose the difference between stored data and derived data
- Don't expose implementation details of a class
- Don't expose a class's internal structure

class Good

```hpp
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <vector>
#include <optional>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

private:
    std::optional<int> firstnum;
    std::vector<int> numbers;
};

#endif
```

```cpp
/*
    Good.cpp

    Implementation of class Good
*/

#include "Good.hpp"

// constructor
Good::Good()
    : numbers(1000)
{}

// operation
void Good::op(int n) {

    // record first number inserted
    if (!firstnum)
        firstnum = n;

    numbers.push_back(n);
}
```

class Good

```
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <vector>
#include <optional>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

private:
    std::optional<int> firstnum;
    std::vector<int> numbers;
};

#endif
```

- Actually, is a "good class"
- Follows encapsulation and information-hiding rules, or does it?
- The client code for Good Class:

Must `#include "Good.hpp"`

Knows data member types

`Good.hpp` may depend on a large number of direct and indirect include files for the data members

Client recompiled whenever a change in class Good:

```cpp
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <vector>
#include <optional>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

private:
    std::optional<int> firstnum;
    std::vector<int> numbers;
};

#endif
```

- Add a data member
- Delete a data member
- Change the type of a data member
- Update the include files for a data member
- Even if there is no change in the interface

PIMPL

Pointer to IMPLementation

- AKA, *opaque pointer*, *d-pointer*, *compiler firewall*, *Cheshire Cat*

- Use a pointer to a class/struct declaration for an implementation class in the target class declaration file (i.e., `.hpp`)

- Define the implementation class in the target class definition file (i.e., `.cpp`)

class Good PIMPL

```hpp
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <vector>
#include <optional>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

private:
    std::optional<int> firstnum;
    std::vector<int> numbers;
};

#endif
```

```hpp
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <memory>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

    // destructor
    ~Good();

private:
    struct GoodImpl;
    std::unique_ptr<GoodImpl> impl;
};

#endif
```

PIMPL Advantages

```
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <memory>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

    // destructor
    ~Good();

private:
    struct GoodImpl;
    std::unique_ptr<GoodImpl> impl;
};

#endif
```

- The class implementation is hidden
- Add a new data member to the (private) structure without violating binary compatibility
- Header file containing the class declaration only needs to include those files required for the class interface rather than for its implementation
- Not a new technique archive_entry declaration, archive_entry definition

class Good PIMPL Implementation

```
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <memory>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

    // destructor
    ~Good();

private:
    struct GoodImpl;
    std::unique_ptr<GoodImpl> impl;
};

#endif
```

```
/*
    Good.cpp

    Implementation of class Good
*/

#include "Good.hpp"
#include <vector>
#include <optional>

struct Good::GoodImpl {
    GoodImpl() : numbers(1000) {}
    std::optional<int> firstnum;
    std::vector<int> numbers;
};

// constructor
Good::Good()
    : impl(new GoodImpl)
{}

// destructor
Good::~Good() = default;

// operation
void Good::op(int n) {

    // record first number inserted
    if (!impl->firstnum)
        impl->firstnum = n;
    impl->numbers.push_back(n);
}
```

PIMPL: Include File

```
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

#include <memory>

class Good {
public:
    // constructor
    Good();

    // operation
    void op(int n);

    // destructor
    ~Good();

private:
    struct GoodImpl;
    std::unique_ptr<GoodImpl> impl;
};

#endif
```

- Implement using a `class` or `struct`. Typically, it is a `struct` since there is no need to keep it private.

- A struct/class *declaration* forms an *incomplete type*

- `std::unique_ptr<>` can work with incomplete types (with one issue)

- `std::unique_ptr<>` used instead of raw pointer for RAII. I.e., don't depend on the destructor of the class to run

PIMPL: Implementation File

```cpp
/*
    Good.cpp

    Implementation of class Good
*/

#include "Good.hpp"
#include <vector>
#include <optional>

struct Good::GoodImpl {
    GoodImpl() : numbers(1000) {}
    std::optional<int> firstnum;
    std::vector<int> numbers;
};

// constructor
Good::Good()
    : impl(new GoodImpl)
{}

// destructor
Good::~Good() = default;

// operation
void Good::op(int n) {

    // record first number inserted
    if (!impl->firstnum)
        impl->firstnum = n;

    impl->numbers.push_back(n);
}
```

- Define the PIMPL struct in the definition file
- Target class data member initialization is now done in the PIMPL struct
- Access to "members" is through a pointer
- If you have a default destructor, use the keyword `default` with the destructor definition in the .cpp file so that `std::unique_ptr<>` knows how to delete the incomplete type.

class Stack PIMPL

```hpp
/*
    Stack.hpp
*/

#ifndef INCLUDED_STACK_HPP
#define INCLUDED_STACK_HPP

#include <memory>

class Stack {
public:
    // constructor
    Stack();

    // push on top
    void push(int);

    // top of stack
    int top() const;

    // pop from top
    void pop();

    // empty
    bool empty() const;

    // destructor
    ~Stack();

private:
    struct StackImpl;
    std::unique_ptr<StackImpl> impl;
};

#endif
```

```cpp
/*
    Stack.cpp
*/

#include "Stack.hpp"
#include <vector>

// PIMPL Stack implementation
struct Stack::StackImpl {
    std::vector<int> v;
};

// constructor
Stack::Stack()
    : impl(new StackImpl)
{}

// push on top
void Stack::push(int value) {
    impl->v.push_back(value);
}

// top of stack
int Stack::top() const {
    return impl->v.back();
}

// pop from top
void Stack::pop() {
    return impl->v.pop_back();
}

// empty
bool Stack::empty() const {
    return impl->v.empty();
}

// destructor
Stack::~Stack() = default;
```

A Step Further

```hpp
/*
    Good.hpp
    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

class Good {
public:

    // operation
    virtual void op(int n) = 0;

    // destructor
    virtual ~Good() = default;
};

#endif
```

```hpp
/*
    GoodImplementation.hpp
    Declaration of class GoodImplementation
*/

#ifndef INCLUDED_GOODIMPLEMENTATION_HPP
#define INCLUDED_GOODIMPLEMENTATION_HPP

#include "Good.hpp"
#include <vector>
#include <optional>

class GoodImplementation : public Good {
public:

    // constructor
    GoodImplementation();
    // operation
    virtual void op(int n);

private:
    std::optional<int> firstnum;
    std::vector<int> numbers;
};

#endif
```

A Step Further

```
/*
    Good.hpp

    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

class Good {
public:

    // operation
    virtual void op(int n) = 0;

    // destructor
    virtual ~Good() = default;
};

#endif
```

- We write most of the client code based on the abstract class Good
- Often, the abstract class is a type we use as a parameter, constructor parameter, or field
- Somewhere else in the code, we choose a particular implementation
- A Framework or Toolkit may be written using the abstract class Good
- Then, we provide the implementation class that the Framework or Toolkit uses

Choices

```hpp
/*
    Good.hpp
    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

class Good {
public:

    // operation
    virtual void op(int n) = 0;

    // destructor
    virtual ~Good() = default;
};

#endif
```

```hpp
/*
    Good.hpp
    Declaration of class Good
*/

#ifndef INCLUDED_GOOD_HPP
#define INCLUDED_GOOD_HPP

class Good {
public:

    // operation
    virtual void op(int n) {}

    // destructor
    virtual ~Good() = default;
};

#endif
```

EndNotes

- Abstractions can have a time efficiency cost

- Abstractions can have a comprehension cost

- Requirements for abstraction can vary over time on a project

- Have to develop the skill to know when is enough