Modern and Lucid C++ Advanced for Professional Programmers

Part 12 – Advanced Library Design

Thomas Corbat / Prof. Peter Sommerlad
Rapperswil, 23.02.2017
HS2017
• PIMPL Idiom

• Hourglass Interfaces
PIMPL Idiom
Opaque Types (Incomplete Types)

- Name known (declared) but not the content (structure)
  - Introduced by a forward declaration

- Can be used for pointers and references
  - but not dereference values without definition (access members)

- C only uses pointers
  - void * is the universally opaque pointer in C

- void * can be cast to any other pointer type

- Validity and avoidance of undefined behavior is left to the programmer

- Sometimes char * is used for memory of a given size (see BoundedBuffer)
Problem: even minor/internal changes in a class’ definition require clients to re-compile

- E.g. changing a type of a private member variable

Compilation “Firewall”

- Allow changes to implementation without the need to re-compile users

It can be used to shield client code from implementation changes, e.g., when you want to provide a binary library as a DLL/shared library for clients and want to be able to update the library without having the client code to be re-compiled

- You must not change header files your client relies upon

Put in the "exported" header file a class consisting of a "Pointer to IMPLementation" plus all public member functions to be used

Read self-study material! ([http://herbsutter.com/gotw/100](http://herbsutter.com/gotw/100))
- All internals and details are exposed to those interacting with class Wizard
- Makes changes hard and will require recompile

```cpp
class Wizard { // all magic details visible
    std::string name;
    MagicWand wand;
    std::vector<Spell> books;
    std::vector<Potion> potions;
    std::string searchForSpell(std::string const & wish);
    Potion mixPotion(std::string const & recipe);
    void castSpell(Spell spell);
    void applyPotion(Potion phial);
public:
    Wizard(std::string name = "Rincewind") :
        name { name }, wand{} {
    }
    std::string doMagic(std::string const & wish);
    //...
};
```
PIMPL with std::shared_ptr<class Impl>

- Minimal header (Wizard.h)
- All details hidden in implementation (see next slide)
- Delegation to Impl (see Wizard::doMagic)

**Wizard.h**

```cpp
class Wizard {
    std::shared_ptr<class WizardImpl> pImpl;

public:
    Wizard(std::string name);
    std::string doMagic(std::string wish);
};
```

**Implementation of Wizard**

```cpp
Wizard::Wizard(std::string name) {
    pImpl = std::make_shared<WizardImpl>(name);
}

std::string Wizard::doMagic(std::string wish) {
    return pImpl->doMagic(wish);
}
```
WizardImpl class declaration (in WizardImpl.cpp)

```cpp
#include "../Wizard.h"
#include "WizardIngredients.h"
#include <vector>
#include <algorithm>

class WizardImpl {
    std::string name;
    MagicWand wand;
    std::vector<Spell> books;
    std::vector<Potion> potions;
    std::string searchForSpell(std::string const &wish);
    Potion mixPotion(std::string const &recipe);
    void castSpell(Spell spell);
    void applyPotion(Potion phial);
public:
    WizardImpl(std::string name="Rincewind") : name{name}, wand{}{}
    std::string doMagic(std::string const &wish);
    //...
};
```
- **WizardImpl implementation**
  - in WizardImpl.cpp
  - Example member function WizardImpl::doMagic

```cpp
std::string WizardImpl::doMagic(std::string const &wish) {
  auto spell = searchForSpell(wish);
  if (!spell.empty()) {
    castSpell(spell);
    return "wootsh";
  }
  auto potion = mixPotion(wish);
  if (!potion.empty()) {
    applyPotion(potion);
    return "zapp";
  }
  throw std::logic_error{"magic failed"};
}
```
PIMPL with std::unique_ptr<class Impl>

- Expected required change?

Wizard.h

```cpp
class Wizard {
    std::shared_ptr<class WizardImpl> pImpl;
public:
    Wizard(std::string name);
    std::string doMagic(std::string wish);
};
```

Wizard.h

```cpp
class Wizard {
    std::unique_ptr<class WizardImpl> pImpl;
public:
    Wizard(std::string name);
    std::string doMagic(std::string wish);
};
```

WizardImpl.cpp

```cpp
//Implementation of Wizard
Wizard::Wizard(std::string name):
    pImpl{std::make_shared<WizardImpl>(name)} {
}
```

WizardImpl.cpp

```cpp
//Implementation of Wizard
Wizard::Wizard(std::string name):
    pImpl{std::make_unique<WizardImpl>(name)} {
}
PIMPL with std::unique_ptr<class Impl>

- Won't compile!

std::unique_ptr has 2 template parameters:

  - pointee type
  - deleter for pointee type

The default deleter cannot delete an incomplete type

Compiler says: "NO!"
Definition of implicitly declared Destructor

- [special]/1 states: ... An implicitly-declared special member function is declared at the closing } of the class-specifier.

At this point WizardImpl is incomplete

What can we do?
Define the destructor of Wizard after the definition of WizardImpl

```cpp
class Wizard {
    std::unique_ptr<class WizardImpl> pImpl;
public:
    Wizard(std::string name);
    ~Wizard();
    std::string doMagic(std::string wish);
};
```

```cpp
class WizardImpl {
    //...
};
//...
Wizard::~Wizard() = default;
```
## How should objects be copied?

<table>
<thead>
<tr>
<th>No Copying – Only Moving</th>
<th><code>std::unique_ptr&lt;class Impl&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Declare destructor</td>
</tr>
<tr>
<td></td>
<td>• Move operations =default</td>
</tr>
<tr>
<td>Shallow Copying (Sharing the implementation)</td>
<td><code>std::shared_ptr&lt;class Impl&gt;</code></td>
</tr>
<tr>
<td>Deep Copying (Default for C++)</td>
<td><code>std::unique_ptr&lt;class Impl&gt;</code></td>
</tr>
<tr>
<td></td>
<td>• with DIY copy constructor (use copy constructor of Impl)</td>
</tr>
<tr>
<td></td>
<td>• In the future (&gt; C++17) there might be a <code>std::clone_ptr&lt;&gt;</code></td>
</tr>
</tbody>
</table>

## Can `pImpl` == `nullptr`?

- IMHO: never!

## Can you inherit from PIMPL class?

- Better don’t
Hourglass Interfaces
Library Interfaces

- DLL APIs work best (and cross platform compatible) with C only
  - We ignore the Windows burden of providing DLL-export and DLL-import syntax
- C++ can provide C-compatible function interfaces using `extern "C"` in front of a declaration
- C-APIs are error-prone and can be tedious to use
- C++ exceptions do not pass nicely across a C-API
- Foreign language bindings (e.g. for Python etc) often expect C-APIs
- API - Application Programming Interface
  - If stable, you do not need to change your code, if something changes
- ABI - Application Binary Interface
  - If stable, you can use and share DLLs/shared libraries without recompilation
- Not universally applicable, but very common
- Shape of an hourglass
Let's add some functionality to our Wizard

- doMagic() – still casts a spell ("wootsh") or uses a potion ("zapp")
- learnSpell() – learns a new spell (by name)
- maxAndStorePotion() – creates a potion and puts it to the inventory
- getName() – function to make Java programmers happy, otherwise there wouldn’t be a "getX" function

```cpp
struct Wizard {
    Wizard(std::string name = "Rincewind")
        : name{name}, wand{} {
    }
    char const * doMagic(std::string const & wish);
    void learnSpell(std::string const & newspell);
    void mixAndStorePotion(std::string const & potion);
    char const * getName() const {
        return name.c_str();
    }
};
```
Testing Wizard

- Testing a wizard provides the same view a client has

```cpp
using wizard_client::Wizard;

void canCreateDefaultWizard() {
    Wizard const magician{};
    ASSERT_EQUAL("Rincewind", magician.getName());
}

void canCreateWizardWithName() {
    Wizard const magician{ "Petrosilius Zwackelmann" };
    ASSERT_EQUAL("Petrosilius Zwackelmann", magician.getName());
}

void wizardLearnsSpellAndCanRecall() {
    Wizard magician{};
    magician.learnSpell("Expelliarmus");
    ASSERT_EQUAL("wootsh", magician.doMagic("Expelliarmus"));
}

void wizardMixesPotionAndCanApply() {
    Wizard magician{};
    magician.mixAndStorePotion("Polyjuice Potion");
    ASSERT_EQUAL("zapp", magician.doMagic("Polyjuice Potion"));
}

void unknownMagicFails() {
    Wizard magician{};
    ASSERT_THROWS(magician.doMagic("Expecto Patronum!"), std::runtime_error);
}
```
Abstract data types can be represented by pointers
- Ultimate abstract pointer void *

Member functions map to functions taking the abstract data type pointer as first argument

Requires Factory and Disposal functions to manage object lifetime

Strings can only be represented by char *
- Need to know who will be responsible for memory
- Make sure not to return pointers to temporary objects!

Exceptions do not work across a C API
A Wizard can only be accessed through a pointer (const and non-const)

- Construction and destruction through functions

An error pointer stores messages of exceptions

- Functions that may fail need an error pointer parameter for reporting exceptions
- Errors need to be cleaned up when not used anymore

Member functions take a Wizard (pointer) as first parameter

```c
typedef struct Wizard * wizard;
typedef struct Wizard const * cwizard;
wizard createWizard(char const * name,
                    error_t * out_error);
void disposeWizard(wizard toDispose);

typedef struct Error * error_t;
char const * error_message(error_t error);
void error_dispose(error_t error);

char const *doMagic(wizard w,
                     char const * wish,
                     error_t * out_error);
void learnSpell(wizard w,
                char const * spell);
void mixAndStorePotion(wizard w,
                       char const * potion);
char const *wizardName(cwizard w);
```
What Parts of C++ Can Be Used in an extern "C" Interface?

- Functions, but not templates or variadic
  - No overloading in C!
- C primitive types (char, int, double, void)
- Pointers, including function pointers
- Forward-declared structs
  - Pointers to those are opaque types!
  - Are used for abstract data types
- Enums (unscoped - without class or base type!)
- If using from C must embrace it with extern "C" when compiling it with C++
  - Otherwise names do not match, because of mangling

```
#include <stdio.h>

extern "C" {

    typedef struct Wizard * wizard;
    typedef struct Wizard const * cwizard;
    wizard createWizard(char const * name,
                         error_t * out_error);
    void disposeWizard(wizard toDispose);

    // ... #ifdef __cplusplus } #endif
    #ifdef __cplusplus }
    #endif
```

Wizard.h
Implementing the Opaque Wizard Type

- **Wizard class must be implemented**

- **To allow full C++ including templates, we need to use a "trampolin" class**
  - It wraps the actual Wizard implementation

```
extern "C" {
struct Wizard { // C linkage trampolin
    Wizard(char const * name)
        : wiz{name} {};
    unseen::Wizard wiz;
};

namespace unseen {
struct Wizard {
    // ...
    Wizard(std::string name = "Rincewind")
        : name{name}, wand{} {}
    char const * doMagic(std::string const & wish);
    void learnSpell(std::string const & newspell);
    void mixAndStorePotion(std::string const & potion);
    char const * getName() const {
        return name.c_str();
    }
};
}
```

**Note:** The Hairpoll example of Stefanus Du Toit has non-standard code in the trampolin
Dealing with Exceptions

- Remember the 5 ways to deal with errors!
- You can’t use references in C API, must use pointers to pointers
- In case of an error, allocate error value on the heap
  - You must provide a disposal function to clean up
- You can use C++ types internally (std::string)
- It is safe to return the char const *
  - because caller owns the object providing the memory

```c
typedef struct Error * error_t;
char const * error_message(error_t error);
void error_dispose(error_t error);

wizard createWizard(char const * name, error_t * out_error);
```

```cpp
extern "C" {
struct Error {
    std::string message;
};

const char * error_message(error_t error) {
    return error->message.c_str();
}

void error_dispose(error_t error) {
    delete error;
}
}```
Creating Error Messages from Exceptions

- Call the function body and catch exceptions
- Map them to an Error object
- Set the pointer pointed to by out_error
  - Use pointer to pointer as reference to pointer
  - Passed out_error must not be nullptr!

```cpp
template<typename Fn>
bool translateExceptions(error_t * out_error, Fn && fn) {
    try {
        fn();
        return true;
    }
    catch (const std::exception& e) {
        *out_error = new Error{e.what()};
        return false;
    }
    catch (...) {
        *out_error = new Error{"Unknown internal error"};
        return false;
    }
}

wizard createWizard(const char * name, error_t * out_error) {
    wizard result = nullptr;
    translateExceptions(out_error,[&] {
        result = new Wizard{name};
    });
    return result;
}
```
Client-side C++ usage requires mapping error codes back to exceptions

- Unfortunately exception type doesn’t map through
- But can use a generic standard exception
  - `std::runtime_error`, keep the message
- Dedicated RAII class for disposal

Temporary object with throwing destructor

- Strange but possible
- Automatic type conversion passes the address of its guts (opaque)
- Tricky, take care you don’t leak when creating the object!

```
struct ErrorRAII {
    ErrorRAII(error_t error) : opaque {error} {}
    ~ErrorRAII() {
        if (opaque) {
            error_dispose(opaque);
        }
    }
    error_t opaque;
};

struct ThrowOnError {
    ThrowOnError() = default;
    ~ThrowOnError() noexcept(false) {
        if (error.opaque) {
            throw std::runtime_error{error_message(error.opaque)};
        }
    }
    operator error_t*() {
        return &error.opaque;
    }
private:
    ErrorRAII error{nullptr};
};
```
Using ThrowOnError in Client API

```cpp
struct ThrowOnError {
    ThrowOnError() = default;
    ~ThrowOnError() noexcept(false) {
        if (error.opaque) {
            throw std::runtime_error{error_message(error.opaque)};
        }
    }
    operator error_t*() {
        return &error.opaque;
    }
private:
    ErrorRAII error{nullptr};
};

struct Wizard {
    Wizard(std::string const & who = "Rincewind") :
        wiz {createWizard(who.c_str(), ThrowOnError{})} {
    }
    // C linkage trampolin
};
```
Completing the Client Side

- Here the complete view of the client side Wizard class
- Calls "C" functions from global namespace
  - Namespace prefix needed for synonyms to member functions
- Header-only
  - Inline functions delegating
- Need to take care of passed and returned Pointers, esp. char *
  - Do not pass/return dangling pointers!

```
#include <string>

#include <Rice/WizardClient.h>

struct Wizard {
    Wizard(std::string const &who = "Rincewind")
        : wiz{createWizard(who.c_str(), ThrowOnError{})} {
    }
    ~Wizard() {
        disposeWizard(wiz);
    }
    std::string doMagic(std::string const &wish) {
        return ::doMagic(wiz, wish.c_str(), ThrowOnError{});
    }
    void learnSpell(std::string const &spell) {
        ::learnSpell(wiz, spell.c_str());
    }
    void mixAndStorePotion(std::string const &potion) {
        ::mixAndStorePotion(wiz, potion.c_str());
    }
    char const *getName() const {
        return wizardName(wiz);
    }
    private:
        Wizard(Wizard const &) = delete;
        Wizard & operator=(Wizard const &) = delete;
        wizard wiz;
};
```
Really Hiding DLL Content (Compiler-Dependent)

- With the Gnu compiler (and clang I presume)
  - `-fvisibility=hidden`
  - Can be added to suppress exporting symbols
  - Must mark exported ABI functions with default visibility

- Visibility refers to dynamic library/object file export of symbols
  - Windows: `__declspec(dllexport)`
  - See also hairpoll demo project [https://youtu.be/PVYdHDm0q6Y](https://youtu.be/PVYdHDm0q6Y)
  - For more on gcc visibility (expert-level knowledge):
    see [https://gcc.gnu.org/wiki/Visibility](https://gcc.gnu.org/wiki/Visibility)

```c
#define WIZARD_EXPORT_DLL
__attribute__((visibility("default")))

WIZARD_EXPORT_DLL
char const * error_message(error_t error);
WIZARD_EXPORT_DLL
void error_dispose(error_t error);
WIZARD_EXPORT_DLL
wizard createWizard(char const * name,
                      error_t *out_error);
WIZARD_EXPORT_DLL
void disposeWizard(wizard toDispose);
WIZARD_EXPORT_DLL
char const * doMagic(wizard w,
                      char const * wish,
                      error_t *out_error);
WIZARD_EXPORT_DLL
void learnSpell(wizard w, char const *spell);
WIZARD_EXPORT_DLL
void mixAndStorePotion(wizard w, char const *potion);
WIZARD_EXPORT_DLL
char const * wizardName(cwizard w);
```
Library API and ABI design can be tricky for third party users

- Only really a problem if not in-house or all open source
- Even with open source libraries, re-compiles can be a burden
  - There are just too many compiler options
  - Plus DLL versioning

API stability can be important

- PIMPL idiom helps with avoiding client re-compiles
- Not easily applicable with heavily templated code -> that often is header-only

ABI stability is even more important when delivering DLLs/shared libraries

- Only relevant when not header only
- “C” linkage safe, but crippling - Hourglass-Interfaces allow shielding C++ clients from the crippled ABI
  - Still easy to make mistakes (which we always tried to avoid)