Modernes C++

...für Programmierer

Unit 10: Idioms und Patterns

by

Dr. Günter Kolousek

2. November 2019

Überblick

- RAII
- PIMPL
- …?

RAII

- Ressource Acquisition Is Initialization
- Zweck
  - Garantie, dass Ressource am Ende des Scope freigegeben wird
  - Basic Exception Garantie ($\rightarrow$ *error*$_{handling.pdf}$) sicherstellen
- Umsetzung
  - im Konstruktor intialisieren…
  - im Destruktor freigeben
- siehe `std::string`, `std::lock_guard`, `unique_ptr`,...

Virtual Constructor

- Zweck
  - Anlegen eines Objektes oder einer Kopie ohne den konkreten Typ zu kennen
- Umsetzung $\rightarrow$ `unit09`

CRTP

- Curiously Recurring Template Pattern
- Zweck
  - Basisklasse spezialisieren mit einer Subklasse als Templateargument
    - $\rightarrow$ *static polymorphism*
      um Nachteile des *dynamic polymorphism* zu vermeiden
      nur wenn Typen der Objekt von Compiler bestimmbar
- Umsetzung: nächste Folie

CRTP – 2

```cpp
#include <iostream>
using namespace std;  // crpt.cpp

template <typename Child>
struct Base {
    void interface() {
        static_cast<Child*>(this)->implementation();
    }
};
struct Derived : Base<Derived> {
    void implementation() {
        cout << "derived implementation" << endl;
    }
};

int main() {
    Derived d;
    d.interface();  // Prints "Derived implementation"
}
```

```
derived implementation
```

PIMPL

- pointer to implementation
- Zweck
  - Verringerung der Abhängigkeiten zwischen Modulen
- Umsetzung
  - Verschieben der privaten Implementierung einer Klasse in eine separate Struktur

PIMPL – 2

```cpp
#include <iostream>  // pimpl1.cpp
using namespace std;
class Accumulator {
  public:
    Accumulator(double init=0) : val{init} {}
    double value() { return val; }
    Accumulator& operator+=(double v) {
        val += v;
        return *this;
    }
    void reset() { val = 0; }
  private:
    double val;
};
int main() {
    Accumulator acc;
    acc += 2;
    acc += 40;
    cout << acc.value() << endl;
}
```

PIMPL – 3

- Nachteile?

PIMPL – 3

- Nachteile?
  - Implementierung ist in Headerdatei

PIMPL – 3

- Nachteile?
  - Implementierung ist in Headerdatei
    - selbst wenn Methoden in `.cpp` ausgelagert: Instanzvariablen!
    - verletzt das Prinzip *information hiding*!
  - starke Kopplung $\rightarrow$ Compileabhängigkeiten
- Lösung: PIMPL $\equiv$ Pointer to IMPLementation
  - …ist ein: *idiom*

PIMPL – 4

es fehlen die special member functions!

```
#include <iostream>  // pimpl2.cpp
#include <memory>
using namespace std;
class Accumulator {
  public:
    Accumulator(double init=0) : p{make_unique<Impl>(init)} {}
    double value();
    Accumulator& operator+=(double v);
    void reset();
  private:
    struct Impl;  // forward declaration
    const unique_ptr<Impl> p;
};
```

PIMPL – 5

```
struct Accumulator::Impl {  // in separate .cpp file!!!
    Impl(double v) : val{v} {}
    double val{};
    double value() { return val; }
    void incr(double v) { val += v; }
    void reset() { val = 0; }
};
double Accumulator::value() { return p->value(); }
Accumulator& Accumulator::operator+=(double v) {
        p->incr(v);
        return *this;
}
void Accumulator::reset() { p->reset(); }
int main() {
    Accumulator acc;
    acc += 2;
    acc += 40;
    cout << acc.value() << endl;
}
```

PIMPL – 6

… die special member functions:

```
#include <iostream>  // pimpl3.cpp
#include <memory>
using namespace std;
class Accumulator {
  public:
    Accumulator(double init=0) : p{make_unique<Impl>(init)} {}
    double value();
    Accumulator& operator+=(double v);
    void reset();
    Accumulator(const Accumulator& o)
      : p{make_unique<Impl>(*o.p)} {}
    Accumulator(Accumulator&& o) = default;
    Accumulator& operator=(const Accumulator& o);
    Accumulator& operator=(Accumulator&& o) = default;
    ~Accumulator() = default;
  private:
    struct Impl;  // forward declaration
    const unique_ptr<Impl> p;
};
```

PIMPL – 7

```
struct Accumulator::Impl {  // in separate .cpp file!!!
    Impl(double v) : val{v} {}
    double val{};
    double value() { return val; }
    void incr(double v) { val += v; }
    void reset() { val = 0; }
};

double Accumulator::value() { return p->value(); }
Accumulator& Accumulator::operator+=(double v) {
    p->incr(v);  return *this; }
void Accumulator::reset() { p->reset(); }
Accumulator& Accumulator::operator=(const Accumulator& o) {
    *p = *(o.p);  return *this; }

int main() {
    Accumulator acc;
    acc += 2;  acc += 40;
    cout << acc.value() << endl;  // -> 42
}
```

PIMPL – 8

- Vorteile
  - Information hiding!
  - Kopplung reduziert
    - würde `Accumulator::Impl` spezielle Headerdateien benötigen, sind diese jetzt nicht mehr im Interface von `Accumulator`!
  - Schnelleres Übersetzen durch Reduzierung der Übersetzungsabhängigkeiten: *compile time firewall*
    - $\rightarrow$ nur Linken mit `Accumulator::Impl`
    - $\rightarrow$ keine Headerdateien der Implementierung (nur `<memory>` bei Smart-Pointer)
  - größere Binärkompatibilität $\rightarrow$ Größe von `Accumulator` ändert sich nicht, selbst wenn Änderungen an `Impl`
    - d.h. ABI (Application Binary Interface) stabil
  - *Lazy allocation*: Instanz von `Impl` könnte auch *on demand* angelegt werden

PIMPL – 9

- Nachteile
  - Größe eines Objektes um Größe eines Pointers größer
  - weitere Allokation bzw. Freigabe von Speicher
  - Teil des Objektes immer am Heap
    - nicht aneinandergereiht mit Objekt
  - Performance leidet auf Grund der Indirektion
  - komplizierter zu programmieren
  - Copy-Konstruktor implementieren oder löschen
  - Compiler kann `const` bei Methoden nicht mehr überprüfen