Modernes C++

Compilation Firewall
Exception-Safe Function Calls

Philippe Lieser

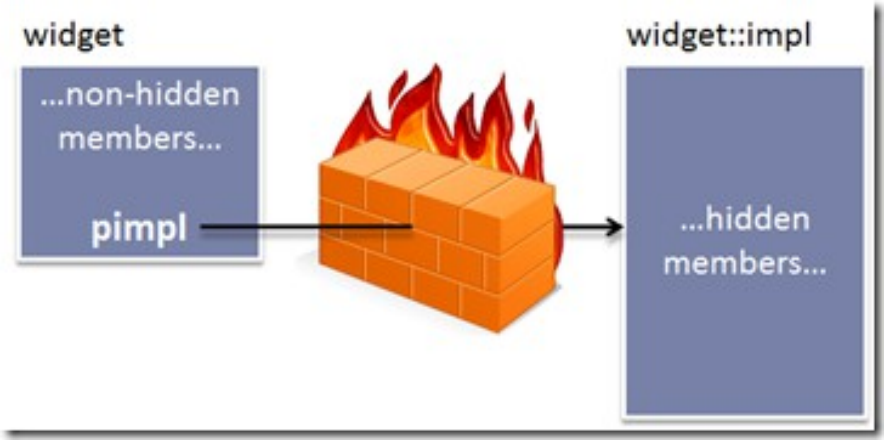
Compilation Firewall

Compilation Firewall - Problem

- wenn im Header File die Klassendefinition geändert wird, muss sämtlicher Code, der diese Benutzt, neu kompiliert werden
- dies gilt auch, wenn nur private member Daten/Funktionen geändert werden
- Gründe:
  - C++ Kompilation basiert auf Text Inklusion
  - Aufrufer wissen Eigenschaften einer Klasse, die von privaten member beeinflusst werden:
    - **Größe und Layout:** Aufrufer muss die Größe und das Layout einer Klasse kennen. Die ist sehr wichtig für Compiler Optimierung.
    - **Funktionen:** *overload resolution* passiert vor *access check*

Compilation Firewall - Lösung

- Implementationsdetails verstecken
- wird durch das Pimpl Idom realisiert



Pimpl Idiom

```
// Pimpl idiom - basic idea
class widget {
    // :::
private:
    struct impl;              // things to be hidden go here
    impl* pimpl_;             // opaque pointer to forward-declared class
};
```

- weniger Compile-Zeit Abhängigkeiten
  - weniger #includes im Header notwendig
  - Änderung an der impl Klasse haben keine Auswirkungen auf die sichtbare Klasse

Pimpl Idiom - Wie am besten in C++ ausdrücken?

```
// in header file
class widget {
public:
    widget();
    ~widget();
private:
    class impl;
    unique_ptr<impl> pimpl;
};

// in implementation file
class widget::impl {
    // :::
};

widget::widget() : pimpl{ new impl{ /*...*/ } } { }
widget::~widget() { }                                  // or =default
```

Wegen user-declared destructor weder kopierbar oder verschiebbar.

Destruktor muss im implementation file definiert werden. Dies liegt daran, dass *unique_ptr* zwar mit einem unvollständigen Typ Instanziiert werden kann, der Destruktor allerdings den vollständigen Typ kennen muss.

Was gehört in die impl-Klasse?

- alle privaten, nicht virtuellen member
- eventuell auch public funktions, die von private funktions gerufen werden
  - um sonst nötigen „back pointer“ zu vermeiden
- wie den back pointer am besten realisieren?
  - *this* als zusätzlichen Parameter übergeben
  - verbraucht nur wenig zusätzlichen Speicher während des Aufrufs
  - der back pointer zeigt so immer auf das richtige Objekt

Pimpl Idiom mittels Hilfsklasse

```
// in header file
#include "pimpl_h.h"
class widget {
    class impl;
    pimpl<impl> m;
    // ...
};

// in implementation file
#include "pimpl_impl.h"
class widget::impl {
    // ...
};
template class
pimpl<widget::impl>;
```

- weniger Code zu schreiben
- dadurch weniger Fehleranfällig
  - z.B. würde bei fehlendem out-of-line destructor erst bei Benutzung der Klasse ein Compiler Fehler entstehen

Wird bei manchen Linkern benötigt.
Mehr dazu in [35.12] und [35.15] aus
http://www.parashift.com/c++-faq-lite/templates.html

pimpl_h.h – Was die Sichtbare Klasse braucht

```
// pimpl_h.h
#ifndef PIMPL_H_H
#define PIMPL_H_H

#include <memory>

template<typename T>
class pimpl {
private:
    std::unique_ptr<T> m;
public:
    pimpl();
    template<typename ...Args> pimpl( Args&& ... );
    ~pimpl();
    T* operator->();
    T& operator*();
};

#endif
```

- Konstruktor mittels Perfect Forwarding

pimpl_impl.h – Was die Implementations Klasse braucht

```
// file pimpl_impl.h
#ifndef PIMPL_IMPL_H
#define PIMPL_IMPL_H

#include <utility>

template<typename T>
pimpl<T>::pimpl() : m{ new T{} } { }

template<typename T>
template<typename ...Args>
pimpl<T>::pimpl( Args&& ...args )
    : m{ new T{ std::forward<Args>(args)... } } { }

template<typename T>
pimpl<T>::~pimpl() { }

template<typename T>
T* pimpl<T>::operator->() { return m.get(); }

template<typename T>
T& pimpl<T>::operator*() { return *m.get(); }

#endif
```

- kümmert sich um die Erzeugung / Zerstörung des impl-Objekts

Exception-Safe Function Calls

Exception-Safe Function Calls (1)

```
// Example 1(a)
//
f( expr1, expr2 );

// Example 1(b)
//
f( g( expr1 ), h( expr2 ) );
```

- Was kann man über die Ausführungsreihenfolge von f, g, h, expr1 und expr2 sagen?
  - 1(a):
    - expr1 und expr2 müssen vor dem Aufruf von f berechnet werden
  - 1(b):
    - expr1 muss vor dem Aufruf von g berechnet werden
    - expr2 muss vor dem Aufruf von h berechnet werden
    - g und h müssen fertig sein, bevor f gerufen werden kann

Exception-Safe Function Calls (2)

```
// Example 2

// In some header file:
void f( T1*, T2* );

// At some call site:
f( new T1, new T2 );
```

- Ist dies exeption safe?
  - Nein!
  - mögliche Ausführungsreihenfolgen:

1. allocate memory for the T1
2. construct the T1
3. allocate memory for the T2
4. construct the T2
5. call f

memory leak von T1, falls 3. oder 4. fehlschlägt

1. allocate memory for the T1
2. allocate memory for the T2
3. construct the T1
4. construct the T2
5. call f

memory leak von T2, falls 3. fehlschlägt
memory leakt von T1, falls 4. fehlschlägt

Exception-Safe Function Calls (3)

```
// Example 3

// In some header file:
void f( std::unique_ptr<T1>, std::unique_ptr<T2> );

// At some call site:
f( std::unique_ptr<T1>{ new T1 }, std::unique_ptr<T2>{ new T2 } );
```

- jetzt exeption safe?
  - immer noch nicht
  - gleiches Problem wie vorher

1. allocate memory for the T1
2. construct the T1
3. allocate memory for the T2
4. construct the T2
5. construct the unique_ptr<T1>
6. construct the unique_ptr<T2>
7. call f

1. allocate memory for the T1
2. allocate memory for the T2
3. construct the T1
4. construct the T2
5. construct the unique_ptr<T1>
6. construct the unique_ptr<T2>
7. call f

Exception-Safe Function Calls (4)

- exeption safety mittels make_unique erreichbar

```
// Example 4

// In some header file:
void f( std::unique_ptr<T1>, std::unique_ptr<T2> );

// At some call site:
f( make_unique<T1>(), make_unique<T2>() );
```

- mögliche Ausführungsreihenfolgen:

1. call make_unique<T1>
2. call make_unique<T2>
3. call f

1. call make_unique<T2>
2. call make_unique<T1>
3. call f

Quellen

- Guru of the Week #100-102 von Herb Sutter
  - http://herbsutter.com/gotw/
- C++ FAQ von Marshall Cline
  - http://www.parashift.com/c++-faq-lite/templates.html
- http://stackoverflow.com/questions/8595471/does-the-gotw-101-solution-actually-solve-anything