C++ Idiome

Detlef Wilkening

http://www.wilkening-online.de

C++ User-Treffen Aachen am 13.02.2014

C++ Idiome

- **Es gibt weit über 100 Idiome in C++**
- **Wir schauen uns hier nur eine kleine subjektive Auswahl an:**
  - Resource Acquisition Is Initialization (RAII)
  - Make-Funktionen
  - Non Virtual Interface (NVI)
  - Virtual Friend Function
  - Class-Layering
    - Basierend auf Curiously Recurring Template Pattern (CRTP)
  - Virtueller Kopier-Konstruktor
    - Mit einem Beispiel, das alle Idioms im Einsatz zeigt

RAII

C++ Idiome

- **Resource Acquisition Is Initialization**
  - RAII
- **Wichtigstes C++ Idiom**
  - Neben Copy&Swap und Parameter-Übergabe
- **Immer, wenn man eine Ressource belegt, dann sollte man ein Objekt damit initialisieren, dass sich um die Ressource "kümmert"**
  - Man nennt dies einen Ressourcen-Manager
  - Und andere Aufgaben darf das Objekt nicht haben
- **"Kümmern" meint - was soll Geschehen beim:**
  - Kopieren
  - Moven
  - Kopier-Zuweisen
  - Move-Zuweisen
  - Zerstören
  - Und evtl. noch beim Swappen

C++ Idiome

- **Beispiele**
  - Smart-Pointer, z.B. aus dem Standard
    - std::unique_ptr
    - std::shared_ptr
    - std::weak_ptr
    - std::auto_ptr - deprecated
  - Dateien
  - Locks, Mutexe
  - Netzwerk-Verbindungen
  - Maus-Cursor
  - usw.

C++ Idiome

```
#include <memory>
#include <string>
using namespace std;

int main()
{
   // Exception-sicher
   unique_ptr<double> pd(new double(3.14));
   shared_ptr<string> ps(new string("C++"));

   // Keine Freigabe notwendig
}
```

C++ Idiome

```
#include <memory>
#include <string>
using namespace std;

int main()
{
   // Noch besser mit Make-Funktionen - siehe gleich
   unique_ptr<double> pd1 = make_unique<double>(3.14);      // C++14
   shared_ptr<string> ps1 = make_shared<string>("C++");

   // Oder mit mit Make-Funktionen und auto
   auto pd2 = make_unique<double>(3.14);      // C++14
   auto ps2 = make_shared<string>("C++");
}
```

C++ Idiome

```
#include <memory>
#include <string>
using namespace std;

int main()
{
   auto up = make_unique<string>("unique");
   auto up2= up;                                  // Compiler-Fehler
                                                  // Unique-Ptr nicht kopierb.

   auto sp = make_shared<string>("shared");
   auto sp2 = sp;                                 // Okay
}                                                 // String wird "geshared"
```

C++ Idiome

- **Beispiel Maus-Cursor**
  - Maus wird während einer Aktion auf "Warten" umgestellt
  - Und nach der Aktion zurück auf den Standard-Cursor (z.B. Pfeil)
  - Beispiel mit der Windows-API
- **Umsetzung**
  - Kann man händisch programmieren
  - Oder mit Ressoucen-Manager
- **Wer schon mal erlebt hat, dass die Maus nicht zurückgesetzt wird**
  - Der weiß, wie wichtig RAII ist
  - Ausserdem wird der Code einfacher
- **Hinweis**
  - Beispiel ist sehr einfach gehalten
    - Und es funktioniert so auch nicht wirklich
    - Vom Prinzip her schon, aber so nicht in der Konsole
  - Ein echtes Maus-Cursor-Feature sollte etwas leistungsfähiger sein

C++ Idiome

```
#include <chrono>
#include <iostream>
#include <thread>
#include <windows.h>
using namespace std;

int main()
{
   cout << "Start" << endl;

   SetCursor(LoadCursor(0, IDC_WAIT));
   this_thread::sleep_for(chrono::seconds(8));
   SetCursor(LoadCursor(0, IDC_ARROW));

   cout << "Ende" << endl;
}
```

C++ Idiome

```
struct Cursor
{
   Cursor() { SetCursor(LoadCursor(0, IDC_WAIT)); }
   ~Cursor() { SetCursor(LoadCursor(0, IDC_ARROW)); }

   Cursor(const Cursor&) = delete;
   Cursor(Cursor&&) = delete;
   Cursor& operator=(const Cursor&) = delete;
   Cursor& operator=(Cursor&&) = delete;
};
```

C++ Idiome

```
int main()
{
   cout << "Start" << endl;

   Cursor waitcursor;
   this_thread::sleep_for(chrono::seconds(8));

   cout << "Ende" << endl;
}
```

C++ Idiome

- **RAII macht das Leben**
  - Einfacher
  - Sicherer
  - Performanter
  - Wartungs-Freundlicher

- **Und wenn man nur fertige RAII-Klassen nutzt**
- **Dann gilt in C++11 die "Regel-der-0" für einen**
  - Noch einfacherer
  - Noch sicherer
  - Noch wartungs-freundlicher
- **Aber das ist ein anderes Thema**

Make-Funktionen

C++ Idiome

- **Bei Klassen-Templates müssen die Typen explizit angegeben werden**
- **Bei Funktions-Templates kann der Compiler die Typen deduzieren**
- **=>**
- **Nutze Funktions-Templates, um Template-Objekte zu erstellen**

C++ Idiome

```
template<class T> class A
{
public:
   A(T);
};


A<int> a;
```

C++ Idiome

```
template<class T> A<T> make_a(T t)
{
   return A<T>(t);
}


A<int> a = make_a(2);          // C++03 – was bringt das?
auto a = make_a(2);            // C++11 – okay, hier hilfreich
```

C++ Idiome

- **Ist in C++03 hilfreich, wenn:**
  - Die Typen schon an anderen Stellen definiert sind
    - Z.B. bei insert in eine Map
  - Das Ergebnis keine Variable wird, sondern der Typ wieder direkt vom Compiler deduziert wird
    - Z.B. als Input in ein Funktions-Template
    - Beispiel "std::ref" oder "std::cref"
  - Manchmal kann man nett damit tricksen
    - Siehe Vergleich über "make_tuple"
  - Das Ergebnis Typen einer Template-Klassen-Hierarchie sein können
    - Zusammen mit const& - siehe gleich
    - Beispiel: "make_scope_guard"
  - In der Make-Funktion noch mehr passieren muß
    - Z.B. doppelstufige Konstruktion
    - Z.B. "std::make_shared"

C++ Idiome

```
// Typen sind schon an anderer Stelle definiert
// Ist natuerlich in C++1 noch einfacher

map<int, bool> m;
m.insert(map<int, bool>::value_type(3, true));
m.insert(make_pair(4, false));
```

C++ Idiome

```
// Der Ergebnis-Typ wird direkt vom Compiler deduziert

string s;
for_each(begin(v), end(v), bind(&A::fct, obj, _1, ref(s)));
```

C++ Idiome

```
// Vergleich von Datums-Zeit-Objekten ueber "make_tuple" bzw. "tie"

lhs.date < rhs.date ||
lhs.date == rhs.date && lhs.time < rhs.time ||
lhs.date == rhs.date && lhs.time == rhs.time && lhs.id < rhs.id .....

make_tuple(lhs.date, lhs.time, lhs.id)
    < make_tuple(rhs.date, rhs.time, rhs.id);

tie(lhs.date, lhs.time, lhs.id) < tie(rhs.date, rhs.time, rhs.id);
```

C++ Idiome

```
// Unterschiedliche Typen einer Klassen-Hierarchie

class Base
{
public:
   virtual ~Base() {}
   virtual void fct() = 0;

protected:
   Base() {}
};
```

C++ Idiome

```
template<class T> class A : public Base
{
public:
   explicit A(const T&) { ... }
   virtual void fct() { ... }
};

template<class T> A<T> make_a(T&& t)        // C++11
{
   return A<T>(forward<T>(t));
}
```

C++ Idiome

```
auto a = make_a(2);           // C++11 okay
A<bool> a = make_a(true);     // C++03 – was bringt es?

// Basis-Klassen-Const-Referenz
const Base& obj = make_a(3.14);     // const ist extrem wichtig
obj.fct();                          // Objekt lebt hier noch
```

- **Wird eine temporäre Variable an eine const L-Value-Referenz gebunden, so lebt die temporäre Variable solange wie die Referenz**
  - Diese Spezial-Regel wurde speziell für diesen Anwendungs-Fall formuliert
  - Damit können Basis-Klassen-Referenzen mit abgeleiteten Objekten initialisiert werden, die automatisch die richtige Lebensdauer haben
    - Ohne dass man das Objekt in eine eigene Variable kopieren muß
    - Ohne dass man dynamische Speicherverwaltung benötigt
  - Make-Funktionen und const& können sehr hilfreich sein

C++ Idiome

- **Im Standard finden wir ganz viele Make-Funktionen:**
  - make_pair
  - make_tuple
  - make_shared
  - make_unique
    - C++14

NVI

C++ Idiome

- **Non Virtual Interface**
  - NVI
  - Kurz: public Funktionen in Klassen sind niemals virtuell
  - Oder: Virtuelle Funktionen sind entweder protected oder private

C++ Idiome

- **Warum?**
  - Sinn von virtuellen Funktionen: werden überschrieben
  - Daher gibt es nicht eine, sondern viele Implementierungen
  - Damit gibt es keinen eindeutigen Ein- oder Ausstiegs-Punkt
    - Schwer zu erweitern für z.B.:
      - Loging
      - Zeit-Messungen
      - Vor- und Nachbedingungen

C++ Idiome

- **Besser**
  - Virtuelle Funktionen protected oder private machen
  - Im Public-Bereich eine inline Funktion
    - Die kann im Fall der Fälle erweitert werden
  - Kein Performance-Overhead

C++ Idiome

```
class A
{
public:
   virtual ~A() {}

   inline void f() const { f_work(); }

protected:
   A() {}
   virtual void f_work() const;
};
```

C++ Idiome

- **Und wenn wir schon mal dabei sind,**
  ein ordentliches Design zu machen...
  - Stellen Sie direkt Before- und After-Funktionen zur Verfügung
    - Die Performance-Einbußen sind meist vernachlässigbar
  - Zumindest wenn Sie ein Framework schreiben
  - Manche Sprachen haben sowas als Sprachmittel
    - Z.B. CLOS oder Dylan-Varianten
  - Viele Frameworks auch
    - Z.B. Ruby-on-Rails
  - Ihre Anwender werden es Ihnen danken

C++ Idiome

```
class base
{
protected:
   base() {}
   virtual bool before_fct() const { return true; }
   virtual void process_fct() const = 0;
   virtual void after_fct() const { }
public:
   virtual ~base() {}

   inline void fct() const
   {
      if (before_fct())
      {
         process_fct();
         after_fct();
      }
   }
};
```

- **Und macht man seine virtuellen Funktionen nun protected oder private?**
  - Auch private Funktionen lassen sich überschreiben
  - "private" definiert in C++ Zugriffsbereiche, keine Sichtbarkeiten
    - Das hat gewisse Konsequenzen, die heute aber nicht Thema sind
  - Funktionieren tut NVI sowohl "protected" als auch "private"
  - Was ist dann der Unterschied?
    - protected
      - Die überschriebenen Funktionen können die Original-Funktion aufrufen
    - private
      - Die überschriebenen Funktionen können die Original-Funktion NICHT aufrufen
    - Typische Anwendung
      - "pure virtuell functions" sind private, damit niemand versucht sie aufzurufen

Virtual Friend Function

C++ Idiome

- **Problem**
  - Klassen benötigen häufig Friend-Funktionen
    - Operatoren, die nicht Teil der Klasse sein können
      - Z.B. Ausgabe-Operator
    - Oder Funktionen müssen andere freie Funktionen überladen
    - Z.B. Swap-Funktion
- **Auf der anderen Seite sollten die Funktionen virtuell sein**
  - Abgeleitete Klassen sollen sie überschreiben können
- **Aber Friend-Funktionen können nicht virtuell sein, da sie freie Funktionen sein müssen**

C++ Idiome

- **Lösung**
  - Virtual Friend Function Idiom
  - Friend-Funktion delegiert zu einer virtuellen Funktion
- **Beispiel**
  - Ausgabe-Operator für eine Klassen-Hierarchie

C++ Idiome

```
class Base
{
public:
   virtual ~Base() {}

   friend inline ostream& operator<<(ostream& out, const Base&  b)
   {
      return b.print(out);
   }

protected:
   Base();
   virtual ostream& print(ostream&) const;
};
```

CRTP
Class-Layering

C++ Idiome

- **Curiously Recurring Template Pattern**
  - CRTP
  - Achtung: Barton-Nackman Trick ist kein CRTP
    - Sondern basiert auf CRTP
    - Restricted Template Expansion
- **Ableitung von einer Template-Klasse, und Übergabe der abgeleiteten Klassen als Template-Argument**

C++ Idiome

```
template<class TDerived> class Base
{
};

class Derived : public Base<Derived>
{
};
```

- **CRTP**
  - Ermöglicht spezifische Basis-Klassen
    - Jede Klasse bekommt seine eigene Basis-Klasse
  - Ermöglicht das statische Aufrufen von Funktionen aus abgeleiteten Klassen
    - Entgegen der normalen Sichtbarkeit von Vererbung
    - Bilden Mixins wie z.B. in Ruby nach
  - Ermöglicht manchmal das Ersetzen von dynamischer Polymorphie durch statische Polymorphie für bessere Performance
  - Ermöglicht das Einziehen einer generischen Klassen-Ebene in eine Klassen-Hierarchie, bei der Code auf dem Typ der abgeleiteten Klasse basieren kann
    - Class-Layering
    - Schauen wir uns heute als einzige Anwendung von CRTP an
    - Wird später auch im Beispiel "virtueller Kopier-Konstruktor" genutzt

- **CRTP - Class Layering**
- **Problem**
  - Häufig sind Funktionen in Blatt-Klassen gleich
  - Basis-Klassen Funktion geht aber nicht, da der Code typ-abhängig ist
    - Beispiele:
      - Print-Type-Name - siehe nächste Folien (Primitiv-Beispiel, kann man auch anders lösen)
      - virtueller Kopier-Konstruktor - siehe nächstes Idiom
- **Lösung**
  - Class-Layering
  - Template-Klasse, die sich zwischen Basis- und Blatt-Klasse legt
  - Implementiert die überall gleiche Funktionalität
  - Durch CRTP bekommt es die Typ-Information über die Blatt-Klasse

C++ Idiome

```
struct Base
{
   virtual ~Base() {}
   virtual void print_type_name() const = 0;
};
struct A : Base
{
   virtual void print_type_name() const
   {
      cout << typeid(A).name() << endl;
   }
};
struct B : Base
{
   virtual void print_type_name() const
   {
      cout << typeid(B).name() << endl;
   }
};
```

C++ Idiome

```
struct Base
{
   virtual ~Base() {}
   virtual void print_type_name() const = 0;
};

template<class T> struct PTN : Base
{
   virtual void print_type_name() const
   {
      cout << typeid(T).name() << endl;
   }
};

struct A : PTN<A> {};
struct B : PTN<B> {};
```

- **Typische Probleme von Layern**
  - Basis-Klasse soll nicht feststehen
  - Konstruktoren der Basis-Klassen
    - Wie stellt man diese zur Verfügung?
    - In C++03 ging dies nur mit allen möglichen Member-Template-Konstruktoren
      - Trotzdem nur mäßige Lösung
      - R-Values werden nicht optimal gehandelt
      - Viel Schreibarbeit
    - In C+11
      - Member-Templates mit Perfect-Forwarding und Variadic-Templates
      - Und viel einfacher: vererbte Konstruktoren
  - Welchen Cast nimmt im Template-Layer für "this"
    - Nicht Teil des Primitiv-Beispiels eben - aber in der Praxis sehr relevant
      - Kommt im nächsten Idiom vor
    - static_cast
    - dynamic_cast
    - Oder noch andere?

- **Basis-Klasse soll nicht feststehen**
  - Lösung: auch als Parameter an das Layering-Template übergeben

```
struct Base
{
   virtual ~Base() {}
   virtual void print_type_name() const = 0;
};

template<class T, class B> struct PTN : B
{
   virtual void print_type_name() const
   {
      cout << typeid(T).name() << endl;
   }
};

struct A : PTN<A, Base> {};
```

- **Konstruktoren der Basis-Klassen**
  - Wie stellt man diese zur Verfügung?
  - In C++03 ging dies nur mit allen möglichen Member-Template-Konstruktoren
    - Trotzdem nur mäßige Lösung
    - R-Values werden nicht optimal gehandelt
    - Viel Schreibarbeit
  - In C+11
    - Member-Templates mit Variadic-Templates und Perfect-Forwarding
    - Oder viel einfacher: vererbte Konstruktoren

C++ Idiome

```
struct Base
{
   Base();
   Base(int);
   Base(double);
   Base(bool, const string&);
   Base(string&&);
   virtual ~Base() {}
   virtual void print_type_name() const = 0;
};
template<class T, class B> struct PTN : B
{
    using B::B;
    virtual void print_type_name() const
    {
        cout << typeid(T).name() << endl;
    }
};
```

C++ Idiome

```
struct A : PTN<A, Base>
{
   using PTN<A, Base>::PTN<A, Base>;
};

struct B : PTN<B, Base>
{
   using PTN<B, Base>::PTN<B, Base>;
};
```

- **Welchen Cast nimmt im Template-Layer für "this"**
  - Auswahl
    - static_cast
      - Schnell, aber keine Laufzeit-Überprüfung
      - Funktioniert nicht in allen Situation (MV, cross-casts)
    - dynamic_cast
      - Funktioniert in allen Situationen
      - Langsamer, da mit Laufzeit-Überprüfung
      - Gibt bei Zeigern im Fehler-Fall Null-Zeiger zurück
      - Wirft bei Referenzen im Fehler-Fall "std::bad_cast"
    - boost::polymorphic_downcast
      - Boost Conversion Library, Header <boost/cast.hpp>
      - Entspricht static_cast
      - Im Debug-Modus aber mit Laufzeit-Überprüfung (assert)
      - Arbeitet nur mit Zeigern - keine Referenzen
    - boost::polymorphic_cast
      - Boost Conversion Library, Header <boost/cast.hpp>
      - Entspricht dynamic_cast
      - Wirft aber bei Null-Zeigern eine Exception (std::bad_cast)
      - Arbeitet nur mit Zeigern - keine Referenzen

C++ Idiome

- **Was man nimmt, bleibt einem selber überlassen**
- **Meine typische Auwahl**
  - dynamic_cast
    - Da es immer funktioniert
  - boost::polymorphic_downcast
    - Wenn ich glaube das "dynamic_cast" nicht notwendig ist
      - Keine Mehrfach-Vererbung (MV)
      - Keine seitlichen Casts (cross-casts)
    - Ich mich aber absichern will
      - Und das mache ich eigentlich immer

Virtueller Kopier-Konstruktor

C++ Idiome

- **Problem**
  - Konstruktoren können in C++ nicht virtuell sein
  - Wie erzeugt man Kopien in polymorphen Klassen-Hierarchien?
- **Lösung**
  - Virtueller Kopier-Konstruktor
  - Virtuelle Funktion, die dynamisch eine Kopie erstellt und zurückgibt
- **Beispiel**
  - Natürlich mit RAII, NVI, Class-Layering (mit CRTP) ,
    Virtual-Friend-Function und Make-Funktion… ;-)

C++ Idiome

```
class Figure
{
public:
   virtual inline ~Figure() = 0;

protected:
   Figure() {}

private:
   virtual unique_ptr<Figure> process_clone() const = 0;
   virtual ostream& process_print(ostream&) const = 0;
```

C++ Idiome

```
public:
   inline unique_ptr<Figure> clone() const
   {
      return process_clone();
   }

   friend inline ostream& operator<<(ostream& out, const Figure& f)
   {
      return f.process_print(out);
   }

};

inline Figure::~Figure() {}
```

C++ Idiome

```
template<class T, class Base> struct CopyPrintLayer : Base
{
   using Base::Base;

   virtual unique_ptr<Figure> process_clone() const
   {
      return make_unique<T>(*polymorphic_downcast<const T*>(this));
   }

   virtual ostream& process_print(ostream& out) const
   {
      return out << typeid(T).name();
   }
};
```

C++ Idiome

```
struct Circle : CopyPrintLayer<Circle, Figure>
{
   using
    CopyPrintLayer<Circle, Figure>::CopyPrintLayer<Circle, Figure>;
};

struct Triangle : CopyPrintLayer<Triangle, Figure>
{
   using
    CopyPrintLayer<Triangle, Figure>::CopyPrintLayer<Triangle, Figure>;
};
```

C++ Idiome

```
int main()
{
   auto pc1 = make_unique<Circle>();
   cout << *pc1 << endl;                    // -> Circle
   auto pc2 = pc1->clone();
   cout << *pc2 << endl;                    // -> Circle

   auto pt1 = make_unique<Triangle>();
   cout << *pt1 << endl;                    // -> Triangle
   auto pt2 = pt1->clone();
   cout << *pt2 << endl;                    // -> Triangle
}
```

Fazit

C++ Idiome

- **Idiome bieten viele schöne Lösungen für typische Probleme**
- **Viele Idiome machen das Leben "nur" sicherer**
  - Und ihre Anwendung macht Arbeit
    - Z.B. Non Virtual Interface (NVI)
- **Trotzdem lohnt sich die Anwendung**
- **Oder gerade deshalb lohnt sich die Anwendung**

C++ Idiome

- **Dies war nur eine sehr kleine Auswahl an Idiomen**
- **Es gibt noch viel viel mehr**
- **Schauen Sie mal im Internet, in Büchern**
  - Z.B. Wikipedia Buch "More C++ Idioms"
  - http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms
- **Und vielleicht auch in Ihrem eigenen Code...**
  - Vielleicht bringt uns das einige Lightning- oder mehr Vorträge ein ;-)
  - Wer möchte nächstes Mal was erzählen?

- **Ansonsten**
  - Viel Spaß und Erfolg beim Ausprobieren und Anwenden…

Danke