12. The Barton & Nackmann Trick

Mit der Verwendung parametrisierter Basisklassen wird (Policy-) Funktionalität in der Ableitung verfügbar. Kann auch die Template-Basisklasse etwas (alles?) über die Ableitung erfahren ?

YES: The Curiously Recurring Template Pattern (erstmals veröffentlicht im Buch ‚Scientific and Engineering in C++' der beiden Autoren)

```
template <typename Derived>
class CuriousBase {
   ...
};


class Curious: public CuriousBase<Curious> { // !
};
```

12. The Barton & Nackmann Trick

Beispiel: universelle Objektzählung

```
#ifndef OBJECTCOUNTER_H
#define OBJECTCOUNTER_H
 template <typename CountedType>
 class ObjectCounter {
   static size_t count;  // number of existing objects
  protected:
   ObjectCounter() { ++count; }
   ObjectCounter (ObjectCounter<CountedType> const&)
   { ++count; }
   ~ObjectCounter() { --count; }
  public:
   // return number of existing objects:
   static size_t live()
   { return count; }
 };
```

12. The Barton & Nackmann Trick

```
// initialize counter with zero
 template <typename CountedType>
 size_t ObjectCounter<CountedType>::count = 0;
#endif
```

Was ist zu zählen ?

```
// Anwendungsbeispiel:

#include "ObjectCounter.h"
#include <iostream>

template <typename CharT>
class MyString : public ObjectCounter<MyString<CharT> > {  };

void count() { usind std::cout; using std::endl;
 cout<<"number of MyString<char>: "<<MyString<char>::live()<<endl;
 cout<<"number of MyString<wchar_t>: "<<MyString<wchar_t>::live()<<endl;
```

12. The Barton & Nackmann Trick

```
int main() {
{
    MyString<char> s1, s2;
    MyString<wchar_t> ws;

    count();
}
```

13. Generalized Functors

Ziel: flexible Implementation des Command-Patterns

Generalized functor: „any processing invocation that C++ allows, encapsulated as a typesafe first-class object."

Entkopplung von Command-Invocator und Command-Receiver:
(interface separation, time separation)

Lokal: der Invoker kennt u.U. den Receiver gar nicht und umgekehrt

Temporal: die Konfiguration der Aktion kann (lange) vor dem Aufruf stattfinden

Modal: es ist dem Invoker nicht bekannt, wie die Aktion erbracht wird

13. Generalized Functors

```
// feste Kopplung:
window.Resize(0, 0, 200, 100);

// Command pattern:
Command resizeCmd(
   window,              // the object
   &Window::Resize,     // the action
   0, 0, 200, 100);     // arguments
...
// later on:

resizeCmd.Execute();
```

13. Generalized Functors

C++ callable Entities

… was man alles in C++ aufrufen kann (quasi operator ()):

a. C like Funktionen
b. C-like Zeiger auf Funktionen
c. Referenzen auf Funktionen (de facto const Zeiger auf F.)
d. Functors (Objekte von Klassen mit operator ())
e. Das Ergebnis der Anwendung von operator .* bzw. ->* mit einem Zeiger auf eine Memberfunktion als rechte Seite

Wunsch: das Command-Pattern soll alle Fälle abdecken

Übliche Idee (Basisklasse mit Ableitungen für Varianten) scheitert

13. Generalized Functors

C++ callable Entities

```
void foo();
struct X {
     static void foo();
     void operator()();
     void bar();
} f;
void (*pF)() = & foo;
void (&rF)() = foo;
void (*psF)() = &X::foo;
void (X::*pM)() = &X::bar;
X* p = &f;
```

```
foo();          // a.
(*pF)();        // b.
(*psF)();       // b. !
rF();           // c.
f();            // d.
(f.*pM)();      // e.
(p->*pM)();     // e.
```

13. Generalized Functors

Einschub: `const T&` oder `T const & ?`

[http://stackoverflow.com/questions/2640446/why-do-some-people-prefer-t-const-over-const-t]:

Eine Frage des Geschmacks?

> … one reason for choosing between the two is whether you want to read it like the compiler (right-to-left) or like English (left-to-right). If one reads it like the compiler, then "T const&" reads as "& (reference) const (to a constant) T (of type T)". If one reads it like English, from left-to-right, then "const T&" is read as "a constant object of type T in the form of a reference.

Nicht nur !

> This will make a difference when you have more then one const/volatile modifiers. Then putting it to the left of the type is still valid but will break the consistency of the whole declaratiion. For example:
>
> `T const * const *p;`
> means that p is a pointer to const pointer to const T and you consistenly read from right to left.
>
> `const T * const *p;`
> means the same but the consistency is lost and you have to remember that leftmost const/volatile is bound to T alone and not T *.

13. Generalized Functors

Interessant: es wäre falsch zu sagen, Callable Entities sind Objekte von Typen, für die `operator()` definiert ist !

`f.*pm` und `p->*pm` haben keinen C++ Typ !

Sofern das Command-Pattern selbst `operator()` definiert (als `Execute()`) , kann man Kommandos in Kommandos schachteln!

(GoF: Makro-Command-Pattern)

Problem: sämtliche Zeiger haben keine first-class Semantik: beim Kopieren muss man Eigentümer-Belange und Polymorphie beachten

Lösungsidee: handle-body-idiom, [piml-idiom]:

`Functor + FunctorImpl`

13. Generalized Functors

```
// 1. Versuch:
class Functor {
public:
   void operator()();  // only void ???
   // other members
private:  // implementation goes here
};
// 2. Versuch: variables Resultat:
template <typename ResultType>
class Functor {
public:
    ResultType operator()();  // any result !!!
   // other members
private: // implementation goes here
};
```

13. Generalized Functors

Argumente können in beliebiger Anzahl und mit beliebigen Typen übergeben werden --- es gibt aber keine (Anzahl-)variablen Template-Parameter in C++98 (wohl aber in C++0x ☺)

```
// 2 ½. Versuch: variable Argumente
template <typename ResultType>
class Functor { ...
};
template <typename ResultType, typename Param1>
class Functor { ...
};
...Param1, Param2, ... ParamN // N ~ 12 … 15
// is not allowed in C++98
```

13. Generalized Functors

Typlisten lösen das Problem:

```
// 3. Versuch: variable Argumente
template <typename ResultType, class TList>
class Functor { ...
};

// zum Beispiel:
Functor<double, TYPELIST_2(int, double)> myFunctor;
```

☹ Die Implementation muss alle Fälle explizit behandeln, bis 15 in `loki` fertig vorbereitet

13. Generalized Functors

Wieder mal Template specialization ☺

```
template <typename R, class TList>
class FunctorImpl;

template <typename R>
class FunctorImpl<R, NullType> {
public:
  virtual R operator ()() = 0;
  virtual FunctorImpl* Clone() const = 0;
  virtual ~FunctorImpl() {}
};
```

13. Generalized Functors

```
template <typename R, typename P1>
class FunctorImpl<R, TYPELIST_1(P1)> {
public:
  virtual R operator ()(P1) = 0;
  virtual FunctorImpl* Clone() const = 0;
  virtual ~FunctorImpl() {}
};
template <typename R, typename P1, typename P2>
class FunctorImpl<R, TYPELIST_2(P1, P2)> {
public:
  virtual R operator ()(P1, P2) = 0;
  virtual FunctorImpl* Clone() const = 0;
  virtual ~FunctorImpl() {}
};...
```

13. Generalized Functors

`Functor` ist dann klassisches handle-body idiom:

```
template <typename R, class TList>
class Functor {
public:
  Functor();                  // artifacts that
  Functor(const Functor&);    //   prove value
  Functor& operator= (const Functor&); // semantic
  explicit Functor(std::auto_ptr<Impl> spImpl);
  ... // ‚extension constructor‘
private:
  typedef FunctorImpl<R, TList> Impl;
  std::auto_ptr<Impl> spImpl_;
};
```

13. Generalized Functors

Parametertypnamen:

```
template <typename R, class TList>
class Functor { // as above +
   typedef TList ParamList;
   typedef typename TypeAtNonStrict<TList, 0,
EmptyType>::Result Parm1;
typedef typename TypeAtNonStrict<TList, 1,
EmptyType>::Result Parm2;
   ...
};
// TypeAtNonStrict == TypeAt mit explizitem Typ
// (3. Parameter) falls „index out of range“
```

13. Generalized Functors

Muss `Functor` ebenfalls für alle Parameteranzahlen spezialisiert werden (wie `FunctorImpl`) ?

NEIN: genialer Trick – implementiere alle, aber nur eine Version wird verwendet und auch nur diese ist überhaupt korrekt !

```
template <typename R, class TList = NullType>
class Functor {// as above +
  public:
     R operator()() { return (*spImpl_)(); }
     R operator()(Parm1 p1) { return (*spImpl_)(p1); }
     R operator()(Parm1 p1, Parm2 p2)
        { return (*spImpl_)(p1, p2); }
     ... // up to 15 prepared
};
```

13. Generalized Functors

Ein Beispiel:

```
// define a Functor for int x double -> double
Functor<double, TYPELIST_2(int, double)> func;
// use it correctly:
double result = func(4, 5.6);
// wrong usage:
double result = func();
// operator ()() not found, weil
// FunctorImpl<double, TYPELIST_2(int, double)>
// diesen nicht implementiert, sondern ?
```

13. Generalized Functors

Konkrete **`FunctorImpl`** Implementationen: (wobei wir beliebige callable entities hinterlegen wollen!)

Zuerst functors (callables type d.-- Instanzen von Klassen mit op()) (**`Functor`** ist selbst ein functor !):

```
template <typename R, class TList>
class Functor { // as above +
public:
   template <class Fun>  // member template
   Functor(const Fun& fun);
};
```

13. Generalized Functors

Dazu wird eine Ableitung von `FunctorImpl<R, Tlist>` benutzt, die ein `Fun` aufbewahrt und `op()` an diesen weiterleitet, der Trick von oben wird erneut benutzt, um alle Signaturen abzudecken:

```
template <class ParentFunctor, typename Fun>
class FunctorHandler: public FunctorImpl <
          typename ParentFunctor::ResultType,
          typename ParentFunctor::ParmList > {
public:
   typedef typename ParentFunctor::ResultType ResultType;
   FunctorHandler(const Fun& fun): fun_(fun){}
   FunctorHandler* Clone() const {
      return new FunctorHandler(*this);
   }
   ...
```

13. Generalized Functors

```
   ResultType operator()() { return fun_(); }
   ResultType operator()
   (
      typename ParentFunctor::Parm1 p1
   )                                    { return fun_(p1); }
   ResultType operator()
   (
      typename ParentFunctor::Parm1 p1,
      typename ParentFunctor::Parm2 p2,

   )                                  { return fun_(p1, p2); }
   ...
private:
   Fun fun_;
};
```

13. Generalized Functors

Nun kann der member template Konstruktor trivial implementiert werden:

```
template <typename R, class TList>
   template <class Fun>  // member template
   Functor<R, Tlist>::Functor(const Fun& fun)
   : spImpl_(new FunctorHandler<Functor, Fun>(fun)
   {}
```

13. Generalized Functors

```
// Test drive:
#include "Functor.h"
#include <iostream>
using namespace std; using namespace loki;
struct TestFunctor {
  void operator()(int i, double d)
  {  cout << "TestFunctor::operator()(" << i
     << ", " << d << ") called.\n"; }
};
int main() {
  TestFunctor f;
  Functor<void, TYPELIST_2(int, double)> cmd(f);
  cmd(4, 4.5);
}
```

13. Generalized Functors

Weitere konkrete **`FunctorImpl`** Implementationen: (wobei wir weitere callable entities hinterlegen wollen!)

Warum haben wir mit den functors begonnen und nicht mit den (vermeintlich) leichteren callable entities type a., b., c. ???

Überraschung: Alles ist schon implementiert !

```
void TestFunction(int i, double d) {
   cout << "TestFunction::(" << i
      << ", " << d << ") called.\n"; }
int main() {
   Functor<void, TYPELIST_2(int, double)>
                                    cmd(TestFunction);
   cmd(4, 4.5);
}
```

13. Generalized Functors

Template parameter deduction macht's möglich:

`TestFunction` als Parameter passt nur auf member template Konstruktor, also wird dieser mit `void (&)(int, double)` instantiiert

Der Typ von `fun_` im

```
FunctorHandler<Functor<...>, void(&)(int, double)>
```

ist demzufolge ebenfalls `void (&)(int, double)`

13. Generalized Functors

Der Aufruf von `operator()` am

```
FunctorHandler<Functor<…>, void(&)(int, double)>
```

Wird an `fun_()` weitergereicht, ‚which is legal syntax for invoking a function through a pointer [reference] to function'.

Es bleibt ein Problem (‚It couldn't be perfect, could it?'): Wenn `TestFunction` überladen wird, kann man den Typ des Symbols nicht mehr (ohne Aufruf) ermitteln.

Auswege: (typisierte) Initialisierung oder Cast

13. Generalized Functors

```
// as above TestFunction overloaded with:
void TestFunction(int); // another one
int main() {
   // ambiguous:
   // Functor<void, TYPELIST_2(int, double)>
                                          cmd(TestFunction);
   // but:
   typedef void (*TpFun)(int, double);
   TpFun pF = TestFunction; // better &TestFunction
   Functor<void, TYPELIST_2(int, double)> cmd1(pF);
   cmd1(4, 5.6);
   Functor<void, TYPELIST_2(int, double)>
   cmd2(static_cast<TpFun>(TestFunction));
   cmd2(4, 5.6);
}
```

13. Generalized Functors

Was, wenn Argumente bzw. Ergebnis nicht exakt passen, sondern nur umwandelbar sind ?

Noch eine Überraschung: Alles ist schon implementiert !

```
const char * TestFunction (double, double) {
  static const char buffer [] = "Hello, world!";
  return buffer; // perfectly safe !
}
int main() {
  Functor<std::string, TYPELIST_2(int, int)>
                                  cmd(TestFunction);
  cout << cmd(10, 11).substr(7);
}
```