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);
}
```

13. Generalized Functors

A. Alexandrescu (modern C++ design, p. 115):

The generality and flexibility of **`FunctorHandler`** illustrate the power of code generation. The syntactic replacement of template arguments for their respective parameters is typical of generic programming. Template processing predates compiling, allowing you to operate at source-code level. In object-oriented programming, in contrast, the power comes from late (after compilation) binding of names to values. Thus, object-oriented programming fosters reuse in the form of binary components, whereas generic programming fosters reuse at the source-code level. Because source code is inherently more information rich and higher level than binary code, generic programming allows richer constructs. This richness, however, comes at the expense of lowered runtime dynamism. You cannot do with STL what you can do with CORBA, and vice versa. The two techniques complement each other.

13. Generalized Functors

Es bleiben Zeiger auf Memberfunktionen im *functor* zu hinterlegen:

Syntaktisch anders, u.U. seltener benutzt, daher ein Beispiel (aus mC++)

```
#include <iostream>
using namespace std;
class Parrot { public:
   void Eat() {
      cout<<“Tsk, knick, tsk...\n“;}
   void Speak(){
      cout<<“Oh Captain, my Captain!\n“;}
};
```

13. Generalized Functors

```
int main() {
   typedef void (Parrot::*TpMemFun)();
   TpMemFun pActivity = &Parrot::Eat;
   Parrot geronimo;
   Parrot* pGeronimo = &geronimo;
   (geronimo.*pActivity)(); // eat!
   (pGeronimo->*pActivity)(); // eat!
   pActivity = &Parrot::Speak;
   (geronimo.*pActivity)(); // speak!
}
```

.* und ->* sind zweistellige Operatoren, die ein Ergebnis liefern, welches keinen C++-Typ hat, sondern nur (sofort) als Funktion gerufen werden kann (*unstable fusion of an object and a member function pointer, a curiously half-baked concept in C++*)

13. Generalized Functors

„... it‘t good to keep things generic and not to jump too early into specifity.“

→ Objekttyp [Parrot] und Memberfunktionstyp [(Parrot::*)()] werden zu Template-Parametern:

```
template <
   class ParentFunctor,
   typename PointerToObj,
   typename PointerToMemFn
>
class MemFunHandler
: public FunctorImpl <
      typename ParentFunctor::ResultType,
      typename ParentFunctor::ParmList
> {
public:
   typedef typename ParentFunctor::ResultType ResultType;
```

13. Generalized Functors

```
// MemFunHandler cont.
MemFunHandler(
      const PointerToObj& pObj,
      PointerToMemFn pMemFn
) : pObj_(pObj), pMemFn_(pMemFn){}

MemFunHandler* Clone() const {
   return new MemFunHandler(*this);
}
ResultType operator()() {
   return ((*pObj_).*pMemFn_)();
} // no param
```

13. Generalized Functors

```
// MemFunHandler cont.
ResultType operator()(
   typename ParentFunctor::Parm1 p1
) {
   return ((*pObj_).*pMemFn_)(p1);
} // one param

ResultType operator()(
   typename ParentFunctor::Parm1 p1,
   typename ParentFunctor::Parm2 p2
) {
   return ((*pObj_).*pMemFn_)(p1, p2);
} // two param
// etc. up to ~15 params
```

13. Generalized Functors

```
   // MemFunHandler cont.
private:
   PointerToObj pObj_;
   PointerToMemFn pMemFn_;
}; //  MemFunHandler
```

Warum parametrisiert mit Objektzeigertyp **(PointerToObj)** und nicht mit Objekttyp* ?

```
// etwa:
private: Obj* pObj_;
public: MemFunHandler(Obj* pObj, ...)
        : pObj_(pObj), ...{}
```

13. Generalized Functors

Die erste Lösung ist generischer, weil ein beliebiger (smart) Zeigertyp benutzt werden kann und nicht nur *hard-wired* kodierte C-Zeiger !

```
// test drive:
#include “Functor.h“
#include <iostream>
using namespace std;

class Parrot { /* as above */ };
int main() {
   Parrot geronimo;
   Loki::Functor<void>
      cmd1(&geronimo, &Parrot::Eat),
      cmd2(&geronimo, &Parrot::Speak);
   cmd1(); // geronimo.Eat()
   cmd2(); // geronimo.Speak()
}
```

13. Generalized Functors

Ziel ist damit erreicht: alle *callable entities* ‚unter einem Hut'

Neue Ideen: Konvertierung eines Functors in einem anderen, z.B. *Binding*: geg. Ein zweistelliger Funktor, durch Festhalten eines Argumentes wird daraus ein neuer einstelliger !

```
void f() {
   Functor<void, TYPELIST_2(int, int)>
        cmd1(something);
   // bind the first argument to 10:
   Functor<void, TYPELIST_1(int)>
        cmd2(BindFirst(cmd1, 10));
   cmd2(20); // same as cmd1(10, 20);
   Functor<void> cmd3(BindFirst(cmd2, 30));
   cmd3();
}
```

13. Generalized Functors

…. allows packaging of functions and arguments without requiring glue code ….

z.B. redo-Funktionalität in einem Editor: Nutzer tippt ein ‘a‘:

- führe aus **`Document::InsertChar('a');`**
- Merke einen ‚canned functor‘, der enthält: Zeiger auf Dokument, Zeiger auf Memberfunktion, aktuelles Zeichen,
- Kann später bei redo gerufen werden

Ausgangslage für **`BinderFirst`**: für eine Instantiierung vom Typ **`Functor<R,TList>`** soll **`TList::Head`** an einen festen Wert gebunden werden, es entsteht ein **`Functor<R, TListTail>`**

„This being said, implementing the* `BinderFirst` *is a breeze.“

13. Generalized Functors

```
// alles natürlich (wie immer) mit Templates ☺

template <class Incoming>
class BinderFirst
   : public FunctorImpl<
      typename Incoming::ResultType,
      typename Incoming::ParameterList::Tail
   > {
   typedef Functor<
      typename Incoming:: ResultType,
      typename Incoming::ParameterList::Tail
   > Outgoing;
   typedef typename Incoming::Parm1 Bound;
   typename Incoming::ResultType ResultType;
```

13. Generalized Functors

```
   // BinderFirst cont.
public:
   BinderFirst(const Incoming& fun, Bound bound)
   : fun_(fun), bound_(bound) {}

   BinderFirst* Clone() const {
      return new BinderFirst(*this);
   }

   ResultType operator()() { // no args
      return fun_(bound_);
   }
```

13. Generalized Functors

```
   // BinderFirst cont.
   ResultType operator()
      ( typename Outgoing::Parm1 p1 ) { // one arg
      return fun_(bound_, p1);
   }

   ResultType operator()
      ( typename Outgoing::Parm1 p1,
        typename Outgoing::Parm2 p2 ) { // two args
      return fun_(bound_, p1, p2);
   } // etc. up to ~15
private:
   Incoming fun_;
   Bound bound_
};
```

13. Generalized Functors

Beabsichtigte Benutzung war:

```
Functor<void,TYPELIST_1(int)> co(BindFirst(ci,10));
```

Die Verknüpfung erledigt eine Funktion mit Hilfe einer Traits-Klasse:

```
template <class F>
typename Private::BinderFirstTraits<F>::BoundFunctorType
BindFirst(const F& fun,typename F::Parm1 bound) {
 typedef typename
 Private::BinderFirstTraits<F>::BoundFunctorType Outgoing;
 return
 Outgoing(std::auto_ptr<typename Outgoing::Impl>
    (new BinderFirst<F>(fun, bound)));
}
```

13. Generalized Functors

Binding funktioniert auch mit automatischer Konversion:

```
const char* Fun(int i, int j) {
   cout<<“Fun(“<<i<<“, “<<j<<“) called\n“;
   return “0“;
}

int main() {
   Functor<const char*, TYPELIST_2(char, int)> f1(Fun);
   Functor<std::string, TYPELIST_1(double)> f2(
                                        BindFirst(f1, 10));
   f2(15); // Ausgabe ???
}
```

13. Generalized Functors

Effizienzbetrachtung

A.Alexandrescu:"Usually, premature optimization is not recommended. One reason is that programmers are not good at estimating which parts of the program should be optimized and (most important) which should not. However, library writers are in a different situation. They don't know whether or not their library will be used in a critical part of some application, so they should take their best shot on optimizing."

```
// z.B. in Functor<R, TLIST>
R operator ()(Parm1 p1, Parm2 p2)  // zwei Kopien !!!
{                                  // auch bei
   return (*spImpl_)(p1, p2);      // inlining
}
```

13. Generalized Functors

```
// besser ?
R operator ()(Parm1& p1, Parm2& p2)
{
   return (*spImpl_)(p1, p2);
}

...all looks fine, ... until you try something like:

void foo(std::string&, int);
Functor<void, TYPELIST_2(std::string&, int)> cmd (foo);
...
String s;
cmd(s, 5); // error ! Which one ?
```

13. Generalized Functors

`loki::TypeTraits` helfen weiter:

```
template <typename T>
class TypeTraits {
    private:
        template <class U> struct PointerTraits {
            enum { result = false };
            typedef NullType PointeeType;
        };
        template <class U> struct PointerTraits<U*> {
            enum { result = true };
            typedef U PointeeType;
        };
        template <class U> struct ReferenceTraits {
            enum { result = false };
            typedef U ReferredType;
        };
```

13. Generalized Functors

```
template <class U> struct ReferenceTraits<U&> {
    enum { result = true };
    typedef U ReferredType;
};
…
template <class U> struct UnConst {
    typedef U Result;
    enum { isConst = 0 };
};
template <class U> struct UnConst<const U> {
    typedef U Result;
    enum { isConst = 1 };
};
template <class U> struct UnVolatile {
    typedef U Result;
    enum { isVolatile = 0 };
};
```

13. Generalized Functors

```
        template <class U> struct UnVolatile<volatile U> {
            typedef U Result;
            enum { isVolatile = 1 };
        };
public:
    typedef TYPELIST_4(
            unsigned char, unsigned short int,
            unsigned int, unsigned long int
    ) UnsignedInts;
    typedef TYPELIST_4(
            signed char, signed short int,
            signed int, signed long int
    ) SignedInts;
    typedef TYPELIST_3(bool, char, wchar_t) OtherInts;
    typedef TYPELIST_3(float, double, long double) Floats;
```

13. Generalized Functors

```
    enum { isStdUnsignedInt =
             TL::IndexOf<T, UnsignedInts>::value >= 0 };
    enum { isStdSignedInt =
             TL::IndexOf<T, SignedInts>::value >= 0 };
    enum { isStdIntegral =
             isStdUnsignedInt || isStdSignedInt ||
             TL::IndexOf<T, OtherInts>::value >= 0 };
    enum { isStdFloat =
             TL::IndexOf<T, Floats>::value >= 0 };
    enum { isStdArith = isStdIntegral || isStdFloat };
    typedef Select<
       isStdArith || isPointer || isMemberPointer,
       /* yes */ T,              // kopieren ist billiger
       /* no */ ReferredType&    // Referenzen sind billiger
       >::Result ParameterType;
}
```

13. Generalized Functors

T	TypeTraits<T>::ParameterType
U	U if U is primitive; otherwise, const U&
const U	U if U is primitive; otherwise, const U&
U &	U&
const U &	const U &

13. Generalized Functors

Alle Operatoren von Functors sind daher (in `loki`) nicht auf den Typen, sondern auf deren Traitstypen definiert:

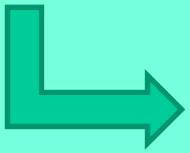
```
// z.B. in Functor<R, TLIST>
R operator ()
(
      typename TypeTraits<Parm1>::ParameterType p1,
      typename TypeTraits<Parm2>::ParameterType p2
)
{
   return (*spImpl_)(p1, p2);
}
```

13. Generalized Functors

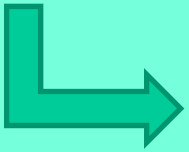
Jetzt Teil der Sprache C++

Der Weg dahin:

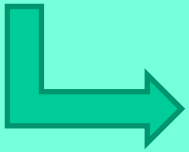
Modern C++ Design (Alexandrescu**, 2001)**



Boost.Bind (Dimov, **~2003)**



std::tr1::bind (ISO**, 2005)**



std::bind (ISO**, 2010)**

folgende Beispiele entstammen Björn Karlsson: "Beyond the C++ Standard Library - An Introduction to Boost"ISBN 0-321-13354-4

[in C++ 0xifizierter Form]

13. Generalized Functors

```
#include <iostream>
#include <functional>

void five_args(int i1, int i2, int i3, int i4, int i5) {
    std::cout<<i1<<i2<<i3<<i4<<i5<<std::endl;
}

int main() {
   int i1=1, i2=2, i3=3, i4=4, i5=5;
   using namespace std::placeholders;
   (std::bind(&five_args, _4,_1,_5,_2,_3))
       (i1,i2,i3,i4,i5);
}
```