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:

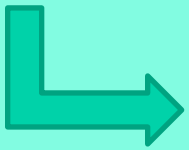
```
// z.B. in Functor<R, TLIST>
R operator ()
(
     typename TypeTraits<Parm1>::ParameterType p1,
     typename TypeTraits<Parm2>::ParameterType p2
)
{
   return (*spImpl_)(p1, p2);
}
```

13. Generalized Functors

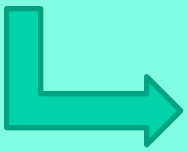
Jetzt Teil der Sprache C++

Der Weg dahin:

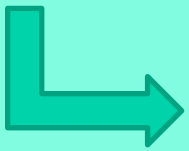
Modern C++ Design (Alexandrescu**, 2001)**



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);
}
```

13. Generalized Functors

```
#include <iostream>
#include <functional>
#include <string>
#include <vector>
#include <algorithm>

class status {
   std::string name_;  bool ok_;
public:
   status(const std::string& name): name_(name), ok_(true) {}
   void break_it() { ok_=false; }
   bool is_broken() const { return !ok_; }
   void report() { std::cout<<name_<<" is "<<
      (ok_ ? "working normally":"terribly broken")<<std::endl;
    }
};
```

13. Generalized Functors

```
int main(){
   std::vector<status> stats;
   stats.push_back(status("status_1"));
 … 2, 3, 4
   stats[1].break_it(); stats[3].break_it();
   // 1: bad multiple end calls
   for(std::vector<status>::iterator it=stats.begin();
        it!=stats.end(); ++it)  it->report();
   // 2: better with for_each, but how?
   std::for_each(stats.begin(), stats.end(),
        std::mem_fun_ref(&status::report) ); // ???
   // 3: even better & clearer & easier to remeber
   namespace PH = std::placeholders;
   std::for_each(stats.begin(), stats.end(),
        std::bind(&status::report, PH::_1) );
}
```

13. Generalized Functors

```
int main(){ // context as before
   std::vector<status*> stats;
   … 2, 3, 4
   stats[1]->break_it();   stats[3]->break_it();


   std::for_each(stats.begin(), stats.end(),
       std::mem_fun(&status::report) );
   // not compiling with std::mem_fun_ref(&status::report)

   // but still ok with the same bind
   namespace PH = std::placeholders;
   std::for_each(stats.begin(), stats.end(),
       std::bind(&status::report, PH::_1) );
}
```

13. Generalized Functors

```
int main(){// context as before
   typedef std::shared_ptr<status> Sptr;
   std::vector<Sptr> stats;
   stats.push_back(Sptr(new status("status_1")));
   … 2, 3, 4

   stats[1]->break_it(); stats[3]->break_it();


   // no way with std::mem_fun_ref and std::_mem_fun
   // but still ok with the same bind
   namespace PH = std::placeholders;
   std::for_each(stats.begin(), stats.end(),
       std::bind(&status::report, PH::_1) );
}
```

13. Generalized Functors

ISO/IEC JTC1 SC22 WG21 N3092

Date: 2010-03-26

ISO/IEC IS 14882

ISO/IEC JTC1 SC22

```
// functional:
namespace std {
…
  template<class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);
  template<class R, class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);
…
} // then, how to store a bound command ?
```

13. Generalized Functors

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

class base {
public: virtual void print() { std::cout<<"I am base.\n"; } };

class derived: public base {
public: virtual void print() {std::cout<<"I am derived.\n"; } };

int main() {
   base b;   derived d;
   auto f = std::bind(&base::print, std::placeholders::_1);
   f(b); f(d);
   base* p = &b;       f(*p);
         p = &d;       f(*p);
}
```

13. Generalized Functors

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

class base { /* as above */ };
class derived : public base { /* as above */ };

template <class Bound>
class F {
public:
   void operator()(base& b) {
        bound_(b);
   }
   F(Bound&& bind): bound_(bind){} // move it !
private:
   Bound bound_;
};
```

13. Generalized Functors

```
// and now instantiating with the unknown:


int main() {
   base b;
   derived d;
   F<decltype(bind(&base::print, std::placeholders::_1))>
            f(bind(&base::print, std::placeholders::_1));
   f(b);
   f(d);

   base* p = &b; f(*p);
         p = &d; f(*p);
}
```