6. ‚Partial Function Specialization‘

Partielle Spezialisierung (wie bei Klassen)
gibt es leider in C++ nicht für Funktionen ☹

Wird dennoch manchmal gebraucht!

```
template <class U, class X>
U* Create (const X& arg)
{ return new U(arg); }
// ein X für ein U vormachen ☺
```

6. ‚Partial Function Specialization‘

Sei **`Widget`** eine fertige (*legacy code*) Klasse bei deren Konstruktion immer ein 2. Parameter (-1) anzugeben ist:

```
// Illegal code – don‘t try this at home
template <class X>
Widget* Create<Widget,X> (const X& arg)
{ return new Widget(arg, -1); }

template <class X>
Widget* CreateWidget (const X& arg)
{ return new Widget(arg, -1); }
// keine Lösung: kein universelles Interface mehr
//          nicht für generischen Code verwendbar ☹
```

6. ‚Partial Function Specialization'

Es bleibt als Ausweg Überladung:

```
template <class U, class X>
U* Create(const X& arg, U /* dummy */)
{ return new U(arg); }

template <class X>
Widget* Create(const X& arg, Widget /* dummy */)
{ return new Widget(arg, -1); }
```

Übergabe des dummy-Arguments ist u.U. aufwendig oder unmöglich ...

6. ‚Partial Function Specialization'

Idee (von Alexandrescu):

nicht die Typen selbst, sondern leichtgewichtige Stellvertreter-(Typen) zur Überladung verwenden

1-1-deutige Abb. von Typen auf Typen: einfach & genial

```
template <typename T>
struct Type2Type {
      typedef T OriginalType;
};
```

T1 !=T2
gdw
Type2Type<T1>!= Type2Type<T2>

6. ‚Partial Function Specialization‘

```
template <class U, class X>
U* Create(const X& arg, Type2Type<U>)
{ return new U(arg); }

template <class X>
Widget* Create(const X& arg, Type2Type<Widget>)
{ return new Widget(arg, -1); }

// using Create:
String* pStr = Create(“Hello“, Type2Type<String>() );
Widget* pWid = Create(100, Type2Type<Widget>() );
```

7. Type Selection

Sometimes generic code needs to select one type or another, depending on a (compile time) boolean constant.

Szenario: Der **`NiftyContainer`** (s.o.) soll als *back-end storage* einen **`std::vector`** verwenden !

`std::vector` hat *value*-Semantik, d.h. es werden Kopien der Elemente verwaltet, ist für polymorphe Typen ungeeignet:

```
class Shape {... virtual void draw() = 0;};
class Circle: public Shape  { ... } kreis;
class Box: public Shape  { ... };

NiftyContainer<Shape> n;
n.add (kreis); // back_end.push_back(Shape-Slice-of-Circle)
```

7. Type Selection

Kopien sind u.U. sogar verboten, *‚if T has disabled its copy constructor (by making it private)* ***as a well-behaved polymorphic class should‘***

Stattdessen sollten Objekte polymorpher Typen im Container als (polymorphe) Zeiger aufbewahrt werden!

```
template <typename T, bool isPoly>
class NiftyContainer {
 // NOT C++:
 isPoly==false:
      std::vector<T> back_end;
 isPoly==true:
      std::vector<T*> back_end;
}
```

7. Type Selection

1. **Lösung: *Type Traits* – ein beigeordneter Typ, in dem die Information hinterlegt wird:**

```
template <typename T, bool isPoly>
class NiftyContainerValueTraits {
    typedef T* ValueType;
};
template <typename T>
class NiftyContainerValueTraits<T, false> {
    typedef T ValueType;
};

template <typename T, bool isPoly>
class NiftyContainer {
   typedef NiftyContainerValueTraits<T, isPoly> Traits;
   typedef typename Traits::ValueType ValueType;
   std::vector<ValueType> back_end;
}
// clumsy and not scaleable
```

7. Type Selection

2. Lösung: ::Loki::Select – *type selection on the spot*

```
template <bool flag, typename T, typename U>
struct Select {
    typedef T Result; // IF (flag) ...
};

template <typename T, typename U>
struct Select<false,T,U> {
    typedef U Result; // IF (!flag) ...
};


template <typename T, bool isPoly>
class NiftyContainer {
typedef typename Select<isPoly, T*, T>::Result ValueType;
   std::vector<ValueType> back_end;
}
```

7. Type Selection

```
#include "TypeManip.h"

namespace Loki {
     template <int v>
        struct Int2Type         {... value ...};       // v
     template <typename T>
        struct Type2Type        {... OriginalType ...}; // T
     template <bool flag, typename T, typename U>
        struct Select           {... Result ...};      // T or U
     template <typename T, typename U>
        struct IsSameType       {... value ...};       // 0 or 1
     template <class T, class U>
        struct Conversion {... exists, exist2Way, sameType ...};
                                                       // 0 or 1
     template <class T, class U>
        struct SuperSubclass    {... value ...};       // 0 or 1
     template<class T,class U>
        struct SuperSubclassStrict {... value ...};  // 0 or 1
}
```

7. Type Selection

```
// Deprecated: Use SuperSubclass class template instead.
#define SUPERSUBCLASS(T, U) \
    ::Loki::SuperSubclass<T,U>::value

// Deprecated: Use SuperSubclassStrict class template instead.
#define SUPERSUBCLASS_STRICT(T, U) \
    ::Loki::SuperSubclassStrict<T,U>::value
```

8. Useable Typeinfos

Standard C++ definiert den Typ **`std::type_info`**

18.5.1 Class `type_info` **[lib.type.info]**

```
namespace std {
  class type_info {
  public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
  private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
  };
}
```

The class `type_info` describes type information generated by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

8. Useable Typeinfos

`std::type_info` ist quasi ‚unbenutzbar':

- Kopien sind nicht erlaubt: **`type_info`**s kann man nicht speichern (z.B. in Containern)

- Zeiger auf **`type_info`**s kann man speichern, aber Vergleiche sind problematisch:

```
std::type_info * info1 = & typeid (SomeType);
std::type_info * info2 = & typeid (SomeType);
// *info1 == *info2 aber u.U.
// info1 != info2 !!!
```

8. Useable Typeinfos

::Loki::TypeInfo: wrapper um type_info mit

- allen Memberfunktionen von `type_info`
- Wert-Semantik (public Copy-Konstruktor & Zuweisung, wo sind diese ???)
- Vergleiche

```
namespace Loki  {    // #include "LokiTypeInfo.h"
    class TypeInfo {
    public:
        // Constructors
        TypeInfo(); // needed for containers
        TypeInfo(const std::type_info&); // non-explicit
        // Access for the wrapped std::type_info
        const std::type_info& Get() const;
        // Compatibility functions
        bool before(const TypeInfo& rhs) const;
        const char* name() const;
    private:
        const std::type_info* pInfo_;
    };
    // Comparison operators
    inline bool operator==(const TypeInfo& lhs, const TypeInfo& rhs)
                  // dito <, !=, >, <=, >=
}
```

9. Typelists

Manchmal will man mittels Template-Instanzierung Code erzeugen lassen für eine (vorab unbekannte) Anzahl von Typen.

Beispiel:

```
class WidgetFactory {
public:
      virtual Window* CreateWindow() = 0;
      virtual Button* CreateButton() = 0;
      virtual ScrollBar* CreateScrollBar() = 0;
};
```

Ist es möglich eine solche konkrete *Factory* aus einer generischen *abstract factory* zu erstellen, a la

```
typedef AbstractFactory<Window,Button,Scrollbar> WidgetFactory;
```

9. Typelists

Dann wäre sicher die Erzeugung der einzelnen „Produkte“ nach folgendem Muster wünschenswert:

```
template <class T>
T* MakeRedWidget(WidgetFactory& factory) {
        T* pW = factory.Create<T>();
        pW->SetColor(RED);
        return pW;
}
```

Leider ist das NICHT möglich, weil:

1. **Templates können KEINE variable Anzahl von Parametern haben (in C++98, in C++0x gibt es sog. *variadic templates*!)**
2. **Virtuelle Funktionen NICHT Templates sein können !**

Ausweg (nach Alexandrescu): Typen, die Listen von Typen verkörpern !

9. Typelists

Typelisten: einfach & genial ***(LISP (Prolog?) lässt grüßen)***

```
template <class T, class U>
struct Typelist {
  typedef T Head;
  typedef U Tail;
};

// +
namespace TL { // local to Loki !
  ... Typelist algorithms ...
};
```

2 Typen --- Typlisten ???
JA: T und/oder U kann selbst Liste sein !!!

9. Typelists

```
typedef Typelist<
  char,
  Typelist<signed char, unsigned char>
> CharList;
```

Probleme:
Typlisten bestehend aus nur einem Typ (Rekursionsanfang?)
Erkennung des letzten Typs (Rekursionsende?)

Lösung: wo immer ein Typ (ohne Semantik) vorkommt:

```
class NullType {}; // a „non“ type

struct EmptyType{}; // a „dont‘t care“ type
```

9. Typelists

Rekursion (wie in Prolog, Lisp, ...) immer im Listen-Tail

```
typedef Typelist<
        char,
        Typelist<
                signed char,
                Typelist<
                        unsigned char,
                        NullType
                >
        >
> AllCharTypes;
```

Alexandrescu: ***„Let's see now how we can manipulate typelists. (Again, this means*** `Typelist` ***types, not*** `Typelist` ***objects.) Prepare for an interesting journey. From here on we delve into the underground of C++, a world with strange, new rules – the world of compile time programming"***

9. Typelists

Wer mag Lisp-(Prolog- ...) Klammergebirge ?

Typelists are just too LISP-ish to be easy to use. LISP-style constructs are great for LISP programmers, but they don't dovetail nicely with C++.

```
typedef Typelist<signed char,
   Typelist<short int,
      Typelist<int, Typelist<long int, NullType> > > >
SignedIntegrals;
```

Makros helfen:

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2)>
// etc. in loki bis
#define TYPELIST_50(...) ...

typedef TYPELIST_4(signed char,short int, int, long int)
              SignedIntegrals;
```

9. Typelists

Letzter Typ in **SignedIntegrals** ?
SignedIntegrals::Tail::Tail::Head // unhandlich

Wir brauchen Algorithmen und geeignete Zugriffsoperationen:

Was einfaches zu Beginn: Ermittle die Länge einer Typliste zur Compile-Zeit!

```
template <class TList> struct Length;
template <>
struct Length<NullType> {
        enum { value = 0 };
};
template <class T, class U>
struct Length<Typelist<T,U> > {
        enum { value = 1 + Length<U>::value };
};
```

9. Typelists

Kann man rekursive Berechnungen zur Compile-Zeit auch in iterative umwandeln (wie dies bei Run-Time-Rekursion immer möglich ist) ?

NEIN: Alle Compile-Zeit-Werte sind unveränderlich, es gibt keine Zuweisung.

Alexandrescu: „*Although C++ is mostly an imperative language, any compile-time computation must rely on techniques that definitely are reminiscent of pure functional languages – languages that cannot mutate values. Be prepared to recurse heavily.*“

Indizierter Zugriff:

```
template <class TList, unsigned int index>
struct TypeAt;
```

9. Typelists

Indizierter Zugriff:

```
template <class TList, unsigned int index>
struct TypeAt;
```

TypeAt
Inputs: Typelist `TList`, index `i`
Outputs: Inner type `Result`

```
If TList is non-null and i is zero
Then Result is the head of TList.
Else
  If TList is non-null and index i is nonzero
  Then Result is obtained by applying TypeAt to the tail of
       TList and i-1.
  Else out-of-bound access produce compile time error
```

9. Typelists

```
template <class Head, class Tail>
struct TypeAt<Typelist<Head, Tail>, 0>
{
        typedef Head Result;
};

template <class Head, class Tail, unsigned int i>
struct TypeAt<Typelist<Head, Tail>, i>
{
        typedef typename TypeAt<Tail, i-1>::Result Result;
};
// Wow !
```

Out-of-bound access: No specialization
for `TypeAt<NullType, x>` wenn man um `x` daneben greift

9. Typelists

Weitere Operationen:

IndexOf
Inputs: Typelist `TList`, Type `T`
Outputs: Inner compile-time constant value

Ermittelt Index des ersten Auftretens von `T` in `TList`, oder -1 (nicht drin)

Append
Inputs: Typelist `TList`, Type or Typelist `T`
Outputs: Inner type definition `Result`

Hängt T an die Liste an, Ergebnis ist eine neuer Typ!

```
typedef Append<SignedIntegrals,
        TYPELIST_3(float, double, long double)>::Result
  SignedTypes;
```

9. Typelists

Erase

Inputs: Typelist `TList`, Type `T`

Outputs: Inner type definition `Result`

Liefert Liste in der das erste Auftreten von `T` in `TList` entfernt wurde, Ergebnis ist eine neuer Typ!

EraseAll

Inputs: Typelist `TList`, Type `T`

Outputs: Inner type definition `Result`

Liefert Liste in der alle Auftreten von `T` in `TList` entfernt wurde, Ergebnis ist eine neuer Typ!

NoDuplicates

Inputs: Typelist `TList`

Outputs: Inner type definition `Result`

9. Typelists

Replace, ReplaceAll
Inputs: Typelist `TList`, Type `T` (to replace), Type `U` (to replace with)
Outputs: Inner type definition `Result`

DerivedToFront
Inputs: Typelist `TList`
Outputs: Inner type definition `Result`

MostDerived
Inputs: Typelist `TList`, Type `T`
Outputs: Inner type definition `Result`

Der am weitesten von T abgeleitete Typ in der Liste oder T, wenn keine Ableitung enthalten

10. Class Generation with Typelists

Erzeuge aus einem nutzerdefinierten Template Code für alle Typen aus einer Typliste:

```
template <class TList,template <class> class Unit>
class GenScatterHierarchy;

template <class T1, class T2,
            template <class> class Unit>
class GenScatterHierarchy<TypeList<T1, T2>, Unit>
      : public GenScatterHierarchy<T1, Unit>
      , public GenScatterHierarchy<T2, Unit> {
public:
  typedef Typelist<T1, T2> TList;
  typedef GenScatterHierarchy<T1, Unit> LeftBase;
  typedef GenScatterHierarchy<T2, Unit> RightBase;
};
```

10. Class Generation with Typelists

```
template <class AtomicType,
          template <class> class Unit>
class GenScatterHierarchy
          : public Unit<AtomicType> {
public:
  typedef Unit<AtomicType> LeftBase;
};

template <template <class> class Unit>
class GenScatterHierarchy<NullType, Unit>
{
};
```

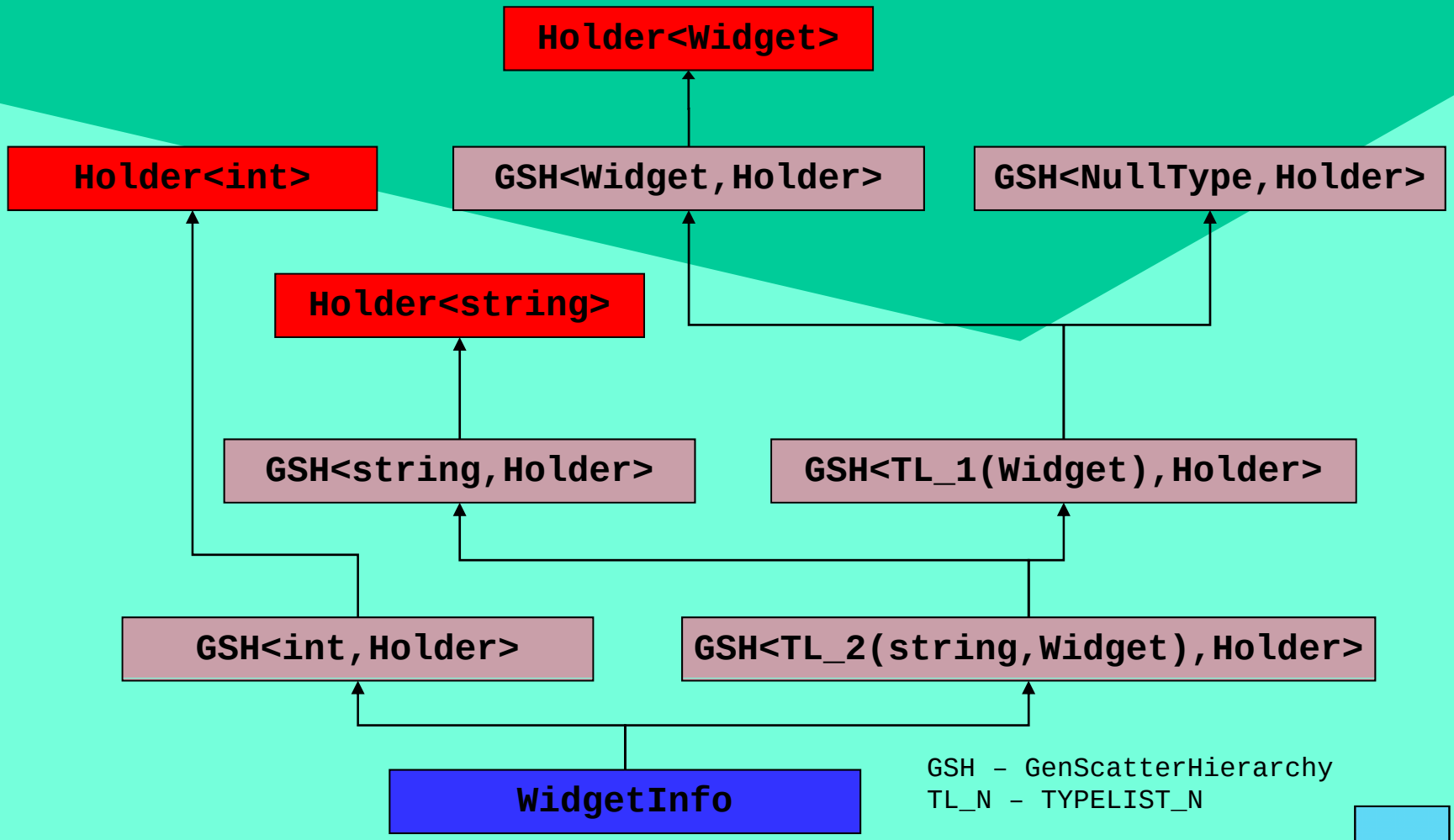
10. Class Generation with Typelists

```
template <class T>
struct Holder {
      T value_;
};

typedef GenScatterHierarchy
      <TYPELIST_3(int,string,Widget), Holder>
      WidgetInfo;

      // Moster-Objekte ?
```

10. Class Generation with Typelists



GSH – GenScatterHierarchy
TL_N – TYPELIST_N

10. Class Generation with Typelists

```
#include <iostream>
#include "Typelist.h"
#include "HierarchyGenerators.h"

using namespace Loki;

template <class T>
struct Holder {
        T value_;
        T foo() {return value_;}
};

struct t {
        int i;
        double d;
        long l;
};
```

10. Class Generation with Typelists

```
typedef TYPELIST_3(int, double, long) t3;

typedef GenScatterHierarchy<t3, Holder> info;

int main() {
      std::cout<<sizeof(info)<<std::endl;
      std::cout<<sizeof(t)<<std::endl;
      Holder<int> *h = new info;
      h->foo();
}
```

10. Class Generation with Typelists

```
mio > g++ -I/vol/home-vol1/simulant/ahrens/loki -o tl tl.cc
1.150u 0.260s 0:04.09 34.4%     0+0k 0+0io 1710pf+0w
mio > ./tl
16
16
mio > ls -l tl
-rwx------    1 ahrens   simulant    12972 2004-05-03 15:54
mio > nm tl | c++filt
...
08048710 W Holder<int>::foo()
...
08048624 T main
...
mio >
```

no monsters around

10. Class Generation with Typelists

Das Wunder von EBCO ☺

EBCO -- Empty **B**ase **C**lass **O**ptimization

1. Auch für leere Klassen gilt sizeof(T) > 0 ! aber

2. Wenn eine leere Klasse als Basisklasse benutzt wird, muss kein Speicherplatz reserviert werden, *„provided that does not cause it to be allocated to the same address as another object or subobject of the same type.“*

```
class Empty {
      typedef int Int;
};
class EmptyToo: public Empty { };
class EmptyThree: public EmptyToo { };
```

10. Class Generation with Typelists

```
int main(){
 using std::cout; using std::endl;
 cout<<"sizeof(Empty):"<<sizeof(Empty)<<endl;
 cout<<"sizeof(EmptyToo):"<<sizeof(EmptyToo)<<endl;
 cout<<"sizeof(EmptyThree):"<<sizeof(EmptyThree)<<endl;
}
```

```
$ ebco
sizeof(Empty):          1
sizeof(EmptyToo):       1
sizeof(EmptyThree):     1
```

10. Class Generation with Typelists

```
// Aber:
class NonEmpty: public Empty, public EmptyToo { };

int main(){
 using std::cout; using std::endl;
 cout<<"sizeof(Empty):"<<sizeof(Empty)<<endl;
 cout<<"sizeof(EmptyToo):"<<sizeof(EmptyToo)<<endl;
 cout<<"sizeof(NonEmpty):"<<sizeof(NonEmpty)<<endl;
}
```

```
$ ebco1
sizeof(Empty):        1
sizeof(EmptyToo):     1
sizeof(NonEmpty):     2
```