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`

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>
{
};
```

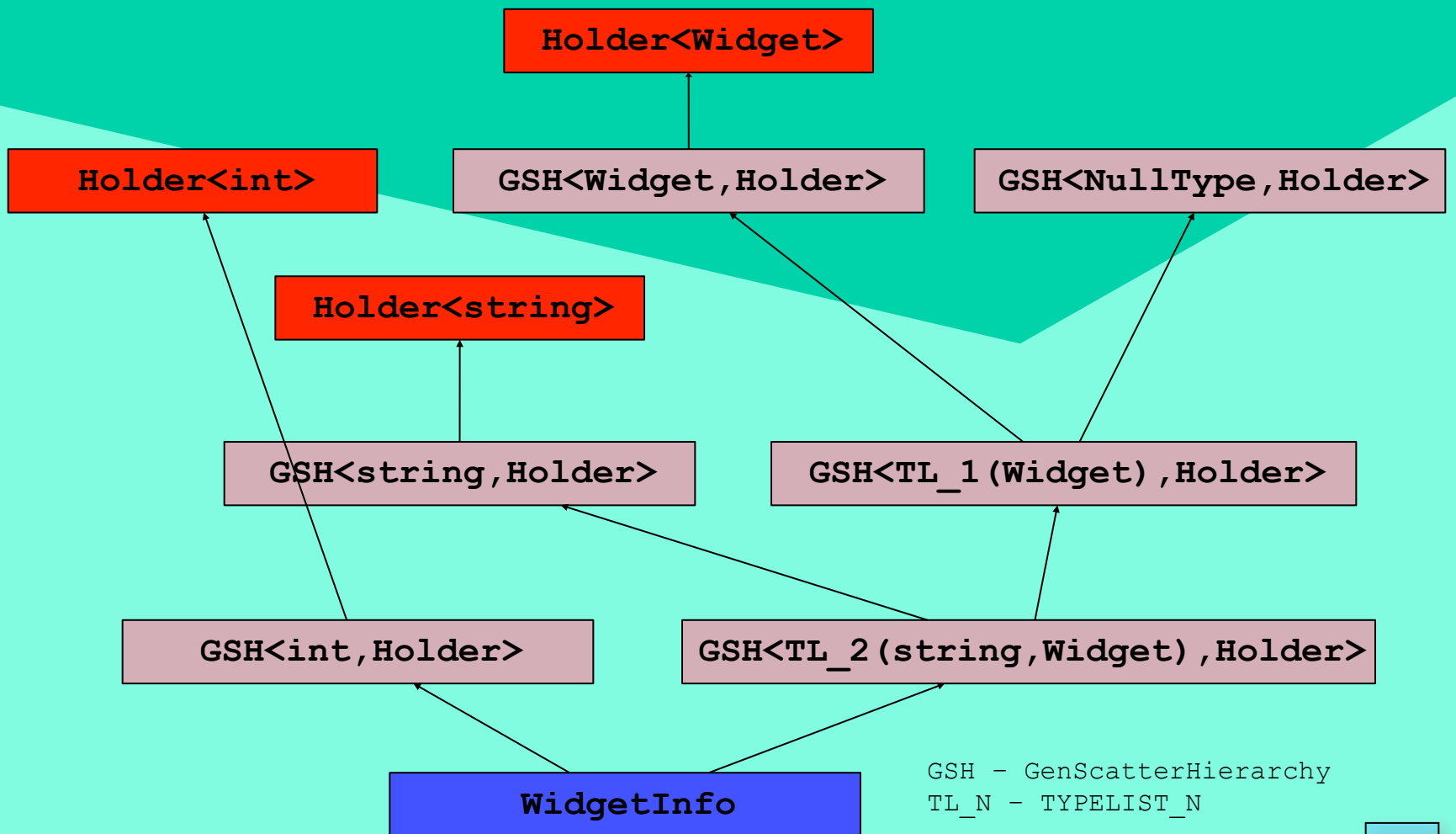
10. Class Generation with Typelists

```
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 Base Class Optimization

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
```

10a. Variadic Templates (C++0x)

```
// variadic.cpp

#include <iostream>

template<typename ... Types>
class simple_tuple;

template<>
class simple_tuple<>
{};
```

10a. Variadic Templates (C++0x)

```
template<typename First, typename ... Rest>
class simple_tuple<First,Rest...>:
    private simple_tuple<Rest...>  {
    First member;
public:
    simple_tuple(First const& f,Rest const& ... rest):
        simple_tuple<Rest...>{rest...}, member{f}   {}

    First const& head() const {
       return member;
    }

    simple_tuple<Rest...> const& rest() const {
        return *this;
    }
```

10a. Variadic Templates (C++0x)

```
template<unsigned index,typename ... Types>
struct simple_tuple_entry;

template<typename First,typename ... Types>
struct simple_tuple_entry<0,First,Types...>
{
    typedef First const& type;

    static type value(simple_tuple<First,Types...> const&
   tuple)
    {
        return tuple.head();
    }
};
```

10a. Variadic Templates (C++0x)

```
template<unsigned index,typename First,typename ... Types>
struct simple_tuple_entry<index,First,Types...> {
    typedef typename simple_tuple_entry<index-1,Types...>::type type;

    static type value(simple_tuple<First,Types...> const& tuple) {
        return simple_tuple_entry<index-1,Types...>::value(tuple.rest());
     }
};

template<unsigned index,typename ... Types>
typename simple_tuple_entry<index,Types...>::type
get_tuple_entry(simple_tuple<Types...> const& tuple) {
    return simple_tuple_entry<index,Types...>::value(tuple);
}
```

10a. Variadic Templates (C++0x)

```
int main()
{
    simple_tuple<int,char,double> st{42,'a',3.141};
    std::cout<<get_tuple_entry<0>(st)<<","
             <<get_tuple_entry<1>(st)<<","
             <<get_tuple_entry<2>(st)<<std::endl;
     std::cout<<"sizeof(st)="<<sizeof(st)<<std::endl;

}
```

```
$ g++ -std=c++0x -v variadic.cpp
… GNU C++ … version 4.4.3 …
$ a.out
42,a,3.141
Sizeof(st)=16
```

11. Policies and Policy Classes

Ziel: implementing

- **safe**
- **efficient**
- **highly customizable**

design elements

Beispiel: eine policy zur Objekterzeugung: new, malloc, cloning prototypes, ...

```
template <class T>
struct OpNewCreator {
      static T* Create() { return new T; }
};
```

11. Policies and Policy Classes

```
template <class T>
struct MallocCreator {
     static T* Create() {
          void* buf = std::malloc(sizeof(T));
          if (!buf) return 0;
          return new (buf) T;
     }
};
```

11. Policies and Policy Classes

```
template <class T>
struct PrototypeCreator {
     PrototypeCreator(T* pObj = 0): proto_(p){}
     T* Create() {
          return proto_ ? proto_ -> Clone() : 0;
     T* getPrototype() { return proto_; }
     void setPrototype(T* pObj){ proto_ = pObj; }
private:
     T* proto_;
};

// ... weitere Policies denkbar
```

11. Policies and Policy Classes

policies sind Klassen, die von anderen Klassen benutzt werden können (Memberdaten, Basis),

policies sind (anders als klassische Interfaces [C++: collections of pure virtual functions]) lose gekoppelt,

policies sind Syntax- (nicht Signatur-) orientiert

Beispiel:

An einer Creator-Policy kann man `Create` rufen und erhält ein neues T-Objekt

11. Policies and Policy Classes

```
// library code with policy based object creation:
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{  ...
}; // host class

// application code:
typedef WidgetManager<OpNewCreator<Widget> > WM;
WM theWidgetManager;
// create a new Widget:
Widget* widget = theWidgetManager.Create();
```

This is the gist of policy-based class design.

11. Policies and Policy Classes

Redundanz: Obwohl ein WidgetManager immer Widgets erzeugt, muss der Typ der Policy mitgeteilt werden!

Besser: template templates

```
// library code with policy based object creation:
template
<template <class Created> class CreationPolicy>
class WidgetManager : public

      CreationPolicy<Widget>
{   ...
}; // host class
```

11. Policies and Policy Classes

```
// application code:
typedef WidgetManager<OpNewCreator> WM;
// library code with policy based object creation:
template
<template <class Created> class CreationPolicy>
class WidgetManager : public

      CreationPolicy<Widget>
{   ...
    // could even create other things:
    Gadget* gadget =
              CreationPolicy<Gadget>().Create();
}; // host class
```

11. Policies and Policy Classes

Vorteile:

flexible Konfiguration beim Klienten

eigene Policies können problemlos integriert werden

Nachteil:

der Klient muss Policies explizit angeben (auch wenn ihm jede Recht ist)

Ausweg: template default parameter

```
template <
   template <class> class CreationPolicy = OpNewCreator
>
class WidgetManager ...
```

11. Policies and Policy Classes

Einfache und erweiterte Policies:

- **Alle Policies unterstützen Kernmethoden (Create)**
- **Manche können mehr anbieten (get/setPrototype)**

Wenn der Klient eine erweiterte Policy benutzt, kann er auch auf deren vollständige Schnittstelle zugreifen!

```
typedef WidgetManager<PrototypeCreator> PWM;
...
Widget* proto = ...;
PWM mgr;
mgr.setPrototype(proto); ....
```

11. Policies and Policy Classes

Der Klient kann sogar optionale Funktionalität implementieren, die nur mit bestimmten Policies funktioniert:

```
template
<template <class> class CreationPolicy>
class WidgetManager : public
  CreationPolicy<Widget>
{  ...
   void SwitchPrototype(Widget* pNewProto) {
     CreationPolicy<Widget>& policy = *this;
     delete policy.getPrototype();
     policy.setPrototype(pNewProto);
   }
};
```

11. Policies and Policy Classes

Der Klient kann sogar optionale Funktionalität implementieren, die nur mit bestimmten Policies funktioniert:

1. **Die vom Klienten gewählte Policy unterstützt Prototypen:** `SwitchPrototype` **kann benutzt werden !**

2. **Die vom Klienten gewählte Policy unterstützt Prototypen nicht,** `SwitchPrototype` **wird aber benutzt: Compile-Time-Error**

3. **Die vom Klienten gewählte Policy unterstützt Prototypen nicht,** `SwitchPrototype` **wird aber nicht benutzt: alles OK !**

11. Policies and Policy Classes

Wenn Policies public Basisklassen sind, kann man u.U. undefined behaviour erzeugen:

```
typedef WidgetManager<PrototypeCreator> PWM;
PWM wm;
PrototypeCreator<Widget>* pC = &wm;// dubious, but legal
delete pC; // undefined behaviour !
```

Virtuelle Destruktoren in Policies ? NEIN !

(Policies sind oft stateless == keine Memberdaten!)

Besser: `protected` NON-`virtual` Destruktoren !

11. Policies and Policy Classes

Policies können frei kombiniert werden, sie sollten dazu aber orthogonal zueinander (komplett unabhängig) sein!

```
template
<
   class T;
   template <class> class CheckingPolicy,
   template <class> class ThreadingModel
>
class SmartPointer;

typedef
SmartPointer<Widget,EnforceNotNull,SingleThreaded>
SafeWidgetPointer;
```