5. Compile Time Dispatch

- Kann man (statisch) prüfen, ob eine Template-Parameterklasse eine bestimmte Funktionalität bereitstellt? JA ☺
- Wie spezifisch können die Anforderungen sein? SEHR ☺
- Was, wenn die Forderung nicht erfüllt wird?
  Ist es möglich, dann auf eine Standard-Variante zurückzugreifen?

Bisher:

Hat eine Typ eine Funktion `Clone`, so kann man diese verwenden.

Hat er sie nicht, gibt es einen statischen Fehler.

Nun: Hat eine Typ eine Funktion `Clone`, so kann man diese verwenden.

Hat er sie nicht, so will man etwas alternatives tun (z.B. ein Duplikat mit dem Copy-Kostruktor erzeugen `new T(*this)`).

5. Compile Time Dispatch

Run-Time dispatch: if-else (switch) ist manchmal nicht möglich!

Szenario: Design eines generischen Containers `NiftyContainer`

```
template <typename T, bool isPolymorphic>
class NiftyContainer {
  ...
  void foo(T* t) { // no valid C++:
     if (isPolymorphic) // has Clone ☺
     {      T* pNewObj = t->Clone(); ... }
     else // has no Clone ☹
     {      T* pNewObj = new T(*t); ... }
};
```

5. Compile Time Dispatch

Obwohl bereits zur Compile-Zeit klar ist, welcher Zweig von if-else betreten wird, müssen beide fehlerfrei übersetzt werden:

Hat `T` kein `Clone`: Fehler im if-Zweig

Hat `T` `Clone` aber keinen `public` Copy-Konstruktor: Fehler im else-Zweig

Alexandrescu: „It would be nice if the compiler didn't bother about compiling code that's dead anyway, but that's not the case."

5. Compile Time Dispatch

```
#include <iostream>
#include "TypeManip.h"
// SUPERSUBCLASS Makro!

using namespace std;

class Cloneable {
public:     virtual Cloneable* Clone() = 0;
};

class Can: public Cloneable {
public:
        Can(){}
        Can* Clone() { std::cout<<"Cloning via Clone\n"; return this; }
};

class Cannot {
public:
        Cannot(){}
        Cannot (const Cannot&) { cout<<"Making a copy\n";}
};
```

Keine ‚echten' Duplikate

5. Compile Time Dispatch

```
template <typename T, bool isPoly>
class NiftyContainer;

template <typename T>
class NiftyContainer<T, true> {
public:
        void foo(T* t) { T* newT = t->Clone(); }
};

template <typename T>
class NiftyContainer<T, false> {
public:
        void foo(T* t) { T* newT = new T(*t); }
};

int main() {
        NiftyContainer<Can,SUPERSUBCLASS(Cloneable, Can)> c1;
        c1.foo(new Can);
        NiftyContainer<Cannot,SUPERSUBCLASS(Cloneable, Cannot)> c2;
        c2.foo(new Cannot);
```

```
$ nifty
Cloning via Clone
Making a copy
```

5. Compile Time Dispatch

Selektion durch Spezialisierung: viel Redundanz (3 Varianten) ☹

Weitere Selektion durch Überladung:

```
template <typename T, bool isPoly>
class NiftyContainer {
public:
     void foo(T* t, ?true? )
     { T* newT = t->Clone(); }
     void foo(T* t, ?false? )
     { T* newT = new T(*t); }
     void foo(T* t) { foo(t, ?isPoly?); }
}; // Überladung erfolgt anhand von Typen!
```

5. Compile Time Dispatch

Überladung erfolgt anhand von Typen!

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

```
template <int v>
struct Int2Type {
      enum { value = v; }
};
```

```
Wert v → Typ Int2Type<v>
Typ Int2Type<v> → Wert Int2Type<v>::value
```

5. Compile Time Dispatch

```
template <typename T, bool isPoly>
class NiftyContainer {
public:
      void foo(T* t, ::Loki::Int2Type<true> )
      { T* newT = t->Clone(); }
      void foo(T* t, ::Loki::Int2Type<false> )
      { T* newT = new T(*t); }
      void foo(T* t)
      { foo(t, ::Loki::Int2Type<isPoly>() ); }
}; // inlining !
```

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
```

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
```

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