2. Compile-Time-Polymorphism

ad 2.

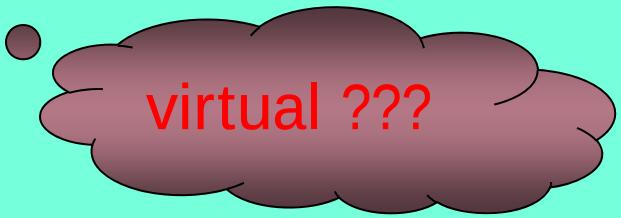
```
// Much better:
#include <fstream>
#include <iostream>
int main(int argc, char* argv[])
{
  using namespace std;
  fstream in, out;
  if (argc > 1)  in.open(argv[1], ios::in  | ios::binary);
  if (argc > 2) out.open(argv[2], ios::out | ios::binary);

  Process
    (in.is_open() ? in : cin, out.is_open() ? out : cout);
}
```

2. Compile-Time-Polymorphism

Wie kann man **`Process`** möglichst flexibel implementieren?

„*In C++, there are four major ways to get polymorphic behavior: virtual functions, templates, overloading and conversions. The first two methods are directly applicable here to express the kind of polymorphism we need.*“

```
// Run-Time-Polymorphism (virtual functions):
void Process
(basic_istream<char>& in, basic_ostream<char>& out)
{
  // ... do something more sophisticated or just plain
  out << in.rdbuf();
}
```



2. Compile-Time-Polymorphism

- ☹ Recht spezifische Typinformation erforderlich (**basic_ios<char>&** geht z.B. nicht) !
- ☹ auf **char** fixiert: was, wenn **wchar_t** (*wide character* == unicode) streams bearbeitet werden sollen ?
- ☹ *late binding* trotz *‚early knowledge‘*

```
// Compile-Time-Polymorphism (templates):
template <typename In, typename Out>
void Process (In& in, Out& out)
{
  // ... do something more sophisticated or just plain
  out << in.rdbuf();
}
```

☺ **Let the compiler deduce the arguments appropriately !**

3. Static Assertions

Ziel:
Prüfung von Eigenschaften (Invarianten, etc.) zur Übersetzungszeit
mit möglichst aufschlussreichen Fehlerausschriften bei Verletzung dieser.

Triviale Varianten:

```
#define STATIC_CHECK(expr) \
     { char unnamed[(expr)? 1 : 0]; }
// it is illegal (in ansi- c++ as well as c) to
// create zero-length arrays
// g++ -Wall –pedantic –ansi:
// warning: ISO C++ forbids zero-size array `unnamed'
```

☹

3. Static Assertions

Triviale Varianten:

```
#define STATIC_CHECK(expr) \
     { int i = 1 / ((expr)? 1 : 0); }
// divide by zero
// g++ -Wall –pedantic –ansi:
// warning: division by zero in `1 / 0‘
```

☹

Besser (wie immer ☺) mit Templates:

```
template <bool> struct CompileTimeError; // no impl.
template <> struct CompileTimeError<true> {};
#define STATIC_CHECK(expr) \
     (CompileTimeError<(expr) != 0>())
```

3. Static Assertions

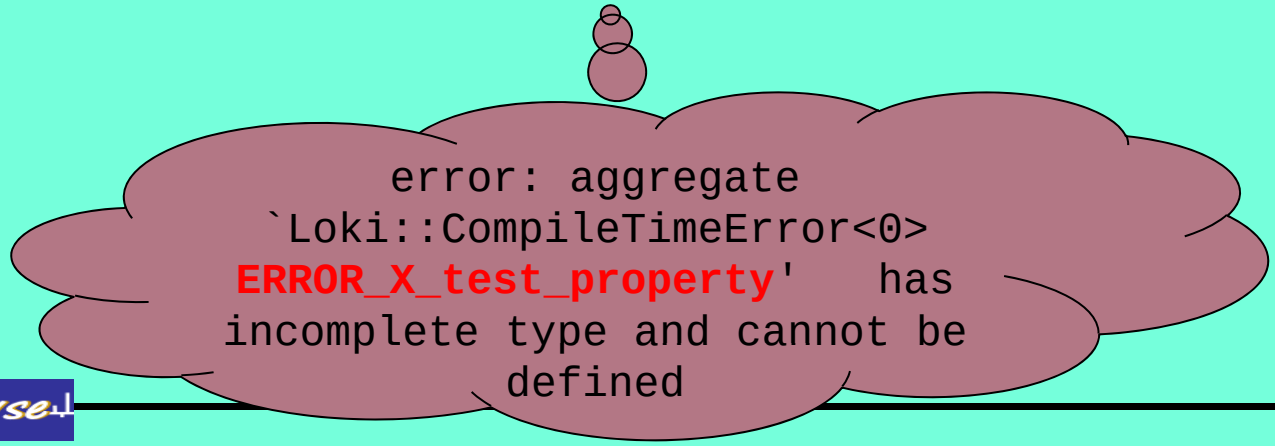
```
////////////////////////////////////////////////////////////////////////////////
// The Loki Library
// Copyright (c) 2001 by Andrei Alexandrescu
// This code accompanies the book:
// Alexandrescu, Andrei. "Modern C++ Design: Generic Programming and Design
//     Patterns Applied". Copyright (c) 2001. Addison-Wesley.
// Permission to use, copy, modify, distribute and sell this software for any
//     purpose is hereby granted without fee, provided that the above copyright
//     notice appear in all copies and that both that copyright notice and this
//     permission notice appear in supporting documentation.
// The author or Addison-Wesley Longman make no representations about the
//     suitability of this software for any purpose. It is provided "as is"
//     without express or implied warranty.
////////////////////////////////////////////////////////////////////////////////
// Last update: June 20, 2001

#ifndef STATIC_CHECK_INC_
#define STATIC_CHECK_INC_

namespace Loki
{
////////////////////////////////////////////////////////////////////////////////
// Helper structure for the STATIC_CHECK macro
////////////////////////////////////////////////////////////////////////////////

    template<int> struct CompileTimeError;
    template<> struct CompileTimeError<true> {};
}
```

3. Static Assertions

```
////////////////////////////////////////////////////////////////////////////////
// macro STATIC_CHECK
// Invocation: STATIC_CHECK(expr, id)
// where:
// expr is a compile-time integral or pointer expression
// id is a C++ identifier that does not need to be defined
// If expr is zero, id will appear in a compile-time error message.
////////////////////////////////////////////////////////////////////////////////

#define STATIC_CHECK(expr, msg) \
    { Loki::CompileTimeError<((expr) != 0)> ERROR_##msg; (void)ERROR_##msg; }


////////////////////////////////////////////////////////////////////////////////
// Change log:
// March 20, 2001: add extra parens to STATIC_CHECK - it looked like a fun
//     definition
// June 20, 2001: ported by Nick Thurn to gcc 2.95.3. Kudos, Nick!!!
////////////////////////////////////////////////////////////////////////////////

#endif // STATIC_CHECK_INC_
```

3. Static Assertions

```
#include “static_assert.h”
// suitable –I option needed

class X {public:
    enum {test=0};
};
int main(){
     STATIC_CHECK(X::test, X_test_property);
}
```



3. Static Assertions

in C++0x Teil der Sprache ☺

```
static_assert(constant_expression,string);

static_assert(sizeof(long) >= 8, "64-bit code generation required.");

struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof(Y), "unexpected padding in S");
```

Nicht zur Prüfung von Laufzeiteigenschaften verwendbar

```
int f(int* p, int n) {
  static_assert(p==0,"p is not null");
  // error: static_assert() expression not a constant expression
}
```

4. Checking template requirements

Motivation: Der **`In`**-Parameter aus **`Process`** muss eine Klasse sein, die eine Methode **`rdbuf()`** bereitstellt!

Allgemeine Fragen:

- Kann man (statisch) prüfen, ob eine Template-Parameterklasse eine bestimmte Funktionalität bereitstellt?
- Wie spezifisch können die Anforderungen sein?
- Was, wenn die Forderung nicht erfüllt wird?

```
// Demonstrationsbeispiel: vgl. Sutter mxc++ Item4
// T must provide  T* T::Clone() const
template <typename T>
class C { /* .... */ };
```

4. Checking template requirements

```
// T must provide  T* T::Clone() const
template <typename T>
class C {
  foo(T* someT) {
      // reicht die bloße Verwendung aus?
      T* t = someT->Clone();
      // ...
  }
};
```

Fall 1: **T** besitzt kein (so aufrufbares) **Clone** – Fehler bei Übersetzung !

Fall 2: **T** besitzt ein (so aufrufbares) **Clone** – die genaue Signatur steht damit noch nicht fest !!!

4. Checking template requirements

Fall 2: **`T`** besitzt ein (so aufrufbares) **`Clone`** – die genaue Signatur steht damit noch nicht fest !!!

- ☺ **`Clone`** kann virtuell sein oder nicht.
- ☹ **`Clone`** kann *default parameter* haben (widerspricht dem *requirement*).
- ☹ **`Clone`** kann non-**`const`** sein (widerspricht dem *requirement*).
- ☹ **`Clone`** kann ein Ergebnis liefern, dass erst in ein **`T*`** umgewandelt wird (widerspricht dem *requirement*).

☹ **Wenn `C::foo` gar nicht aufgerufen wird, ist der Code dennoch richtig: generell werden bei der Template-Instantiierung nur die Methoden erzeugt (und müssen dann übersetzbar sein) die benutzt werden !**

4. Checking template requirements

Man muss den Aufruf offenbar in einer Methode platzieren, die garantiert aufgerufen wird, der Aufruf sollte aber nach Möglichkeit keine Laufzeiteinbußen mit sich bringen (und auch keine Seiteneffekte).

Konstruktor ?

Vermutlich werden Objekte von C benutzt!

Dann aber bitte in allen Konstruktoren ☹

Destruktor !

wird (fast) immer gerufen, außer in Programmen, die sowieso ‚*falsch*' sind (leaks):

```
{  typedef C<SomeType> CC;
   CC* p  = new  CC; // leak ahead
}
```

4. Checking template requirements

```
// T must provide  /*...*/ T::Clone (/*...*/) const
template <typename T>
class C {
public:
        ~C(){
                const T t;
                t.Clone();
        }
};
```

Nur *constness* garantiert, aber Nebeneffekte und Laufzeitbelastung + Forderung nach *default constructable* T

4. Checking template requirements

```
// T must provide  T* T::Clone (void) const
template <typename T>
class C {
public:
      ~C(){
            T* (T::*test)() const = & T::Clone;
            test; // to prevent unused variable
                  // warning, is likely to be
                  // optimized away entirely !
            // ...
      }
};
```

4. Checking template requirements

```
// or even better:
// T must provide  T* T::Clone (void) const
template <typename T>
class C {
        bool ValidateRequirements() const
        {
                T* (T::*test)() const = & T::Clone;
                test;
                // maybe more ...
        }
public:
        ~C(){
                assert (ValidateRequirements());
                // ...
        }
};
```

All traces of the requirement machinery will disappear from release builds !

4. Checking template requirements

```
// Alternative:
// T must provide  T* T::Clone (void) const
template <typename T>
class HasClone {
public:
        static void Constraints() {
                T* (T::*test)() const = & T::Clone;
                test; // maybe more ...
        }
        HasClone() { void (*p)() = Constraints; } // ***
};

template <typename T>
class C: HasClone<T> {
        //...
};
```

//*** publiziert von B. Stroustrup im *C++ Style and Technique FAQ*
nach der ursprünglichen Idee von Alex Stepanov und Jeremy Siek

4. Checking template requirements

Alternative:

Ein Typ **`T`** hat sicher eine Methode **`T* Clone(void) const`** , wenn er eine Ableitung des Typs **`Cloneable`** ist:

```
class Cloneable {
public:
      virtual Cloneable* Clone(void) const = 0;
      // co-variant returt types allow for
      // returning pointers to derived classes
};
```

Frage: Kann man statisch entscheiden ob ein beliebiger Typ D (T) von B (Cloneable) abgeleitet ist?

4. Checking template requirements

1.Methode (nach A. Alexandrescu)

Idee: D ist eine (public) Ableitung von B *genau dann wenn*
ein D* in ein B* umwandelbar ist, D und B verschieden sind und B nicht **void** ist

```
// Conversion T → U
template <typename T, typename U>
class Conversion {
        typedef char Small;
        class Big {char dummy [2];};
        static Small Test(const U&);      // no
        static Big Test(...);             // implementations
        static T makeT();                 // at all
public:
        enum { exists=sizeof(Test(makeT()))==sizeof(Small) };
        enum { sameType = 0 };
};
```

4. Checking template requirements

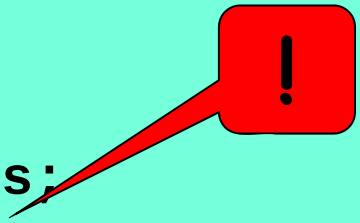
T in U umwandelbar:

Test(T) ist aufzulösen, wegen (impliziter) Umwandlung passt **Test(constU&)** besser als **Test (...)** -> Resultat hat Typ **Small** -> **Conversion<T,U>::exists == 1**

T in nicht U umwandelbar:

Test(T) ist aufzulösen, nur **Test (...)** passt -> Resultat hat Typ **Big** -> **Conversion<T,U>::exists == 0**

```
// include whatever needed
int main() {
  using namespace std;
  cout << Conversion<double,int>::exists;
  cout << Conversion<size_t,vector<int> >::exists;
} // prints: 10 ☺
```



4. Checking template requirements

Typgleichheit ? *Partial Specialization*

```
template <typename T>
class Conversion<T, T> { public:
      enum { exists = 1, sameType = 1 };
};
```

Damit ist alles beisammen:

```
#define SUPERSUBCLASS(B, D) \
      (Conversion<const D*, const B*>::exists &&\
      !Conversion<const B*, const void*>::sameType)
```

4. Checking template requirements

```
#define SUPERSUBCLASS_STRICT (B, D) \
      (SUPERSUBCLASS(B, D) &&\
      !Conversion<const B*, const D*>::sameType)


// H. Sutter mxc++:
template <typename D, typename B>
class IsDerivedFrom {
      class No{};
      class Yes { No no[2]; };

      static Yes Test( B* );      // not defined
      static No  Test( ... );     // not defined
public:
      enum { Is =
         sizeof(Test(static_cast<D*>(0)))==sizeof(Yes) }
};
```

4. Checking template requirements

```
// T must provide  T* T::Clone (void) const
template <typename T>
class C {
      bool ValidateRequirements() const
      {
       typedef IsDerivedFrom<T,Cloneable> Oracle;
       assert (Oracle::Is);
       return true;
      }
public:
      ~C(){
       assert (ValidateRequirements());
       // ...
      }
}; // not really a static test !!!
```

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 !
```