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;
```

12. The Barton & Nackmann Trick

Mit der Verwendung parametrisierter Basisklassen wird (*Policy*-) Funktionalität in der Ableitung verfügbar. Kann auch die Template-Basisklasse etwas (alles?) über die Ableitung erfahren ?

YES: The Curiously Recurring Template Pattern (erstmals veröffentlicht im Buch ‚Scientific and Engineering in C++' der beiden Autoren)

```
template <typename Derived>
class CuriousBase {
   ...
};

class Curious: public CuriousBase<Curious> { // !
};
```

12. The Barton & Nackmann Trick

Beispiel: universelle Objektzählung

```
#ifndef OBJECTCOUNTER_H
#define OBJECTCOUNTER_H
 template <typename CountedType>
 class ObjectCounter {
   static size_t count;  // number of existing objects
  protected:
   ObjectCounter() { ++ObjectCounter<CountedType>::count; }
   ObjectCounter (ObjectCounter<CountedType> const&)
   { ++ObjectCounter<CountedType>::count; }
   ~ObjectCounter() { --ObjectCounter<CountedType>::count; }
  public:
   // return number of existing objects:
   static size_t live()
   { return ObjectCounter<CountedType>::count; }
 };
```

12. The Barton & Nackmann Trick

```
// initialize counter with zero
 template <typename CountedType>
 size_t ObjectCounter<CountedType>::count = 0;
#endif
```

Was ist zu zählen ?

```
// Anwendungsbeispiel:

#include "ObjectCounter.h"
#include <iostream>

template <typename CharT>
class MyString : public ObjectCounter<MyString<CharT> > {  };

void count() { usind std::cout; using std::endl;
 cout<<"number of MyString<char>: "<<MyString<char>::live()<<endl;
 cout<<"number of MyString<wchar_t>: "<<MyString<wchar_t>::live()<<endl;
}
```

12. The Barton & Nackmann Trick

```
int main() {
{
    MyString<char> s1, s2;
    MyString<wchar_t> ws;

    count();
}
```

13. Generalized Functors

Ziel: flexible Implementation des *Command*-Patterns

Generalized functor: *„any processing invocation that C++ allows, encapsulated as a typesafe first-class object.“*

Entkopplung von Command-Invocator und Command-Receiver:
(interface separation, time separation)

Lokal: der Invoker kennt u.U. den Receiver gar nicht und umgekehrt

Temporal: die Konfiguration der Aktion kann (lange) vor dem Aufruf stattfinden

Modal: es ist dem Invoker nicht bekannt, wie die Aktion erbracht wird

13. Generalized Functors

```
// feste Kopplung:
window.Resize(0, 0, 200, 100);

// Command pattern:
Command resizeCmd(
   window,              // the object
   &Window::Resize,     // the action
   0, 0, 200, 100);     // arguments
...
// later on:

resizeCmd.Execute();
```

13. Generalized Functors

C++ callable Entities

... was man alles in C++ aufrufen kann (quasi operator ()):

a. C like Funktionen
b. C-like Zeiger auf Funktionen
c. Referenzen auf Funktionen (de facto const Zeiger auf F.)
d. Functors (Objekte von Klassen mit operator ())
e. Das Ergebnis der Anwendung von operator .* bzw. ->* mit einem Zeiger auf eine Memberfunktion als rechte Seite

Wunsch: das Command-Pattern soll alle Fälle abdecken

Übliche Idee (Basisklasse mit Ableitungen für Varianten) scheitert

13. Generalized Functors

C++ callable Entities

```
void foo();
struct X {
     static void foo();
     void operator()();
     void bar();
} f;
void (*pF)() = & foo;
void (&rF)() = foo;
void (*psF)() = &X::foo;
void (X::*pM)() = &X::bar;
X* p = &f;
```

```
foo();          // a.
(*pF)();        // b.
(*psF)();       // b. !
rF();           // c.
f();            // d.
(f.*pM)();      // e.
(p->*pM)();     // e.
```