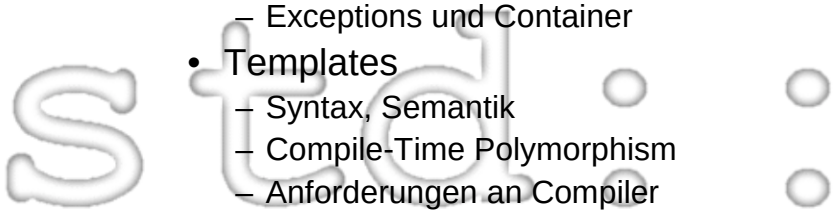
Die C++ Standardbibliothek

- Namespaces
  - Koenig-Lookup
  - namespace std
- Exceptions
  - Standard-Exceptions
  - Exceptions und Container
- Templates
  - Syntax, Semantik
  - Compile-Time Polymorphism
  - Anforderungen an Compiler

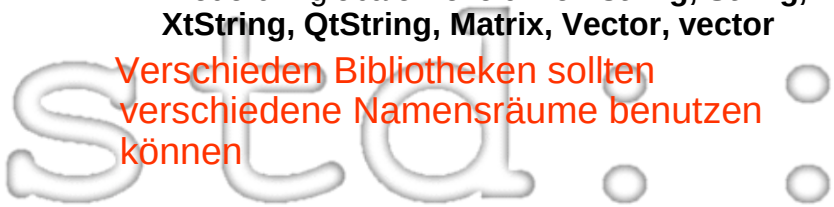


Namespaces in C++

Motivation

- Namenskollision im globalen Namensraum
- Klassen sind ein Hilfsmittel zur Entlastung des globalen Namensraumes,
- Klassennamen sind ihrerseits jedoch (zumeist) wiederum globale Bezeichner: **string, String, XtString, QtString, Matrix, Vector, vector**

Verschieden Bibliotheken sollten verschiedene Namensräume benutzen können



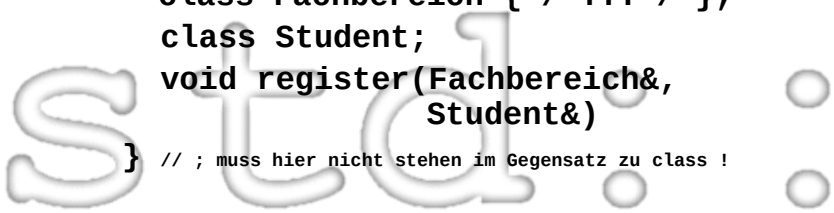
Namespaces in C++

- Präfix sollte durch die Sprache selbst unterstützt werden:

  C++ namespaces

- Syntax: Deklaration wie Klasse (aber keine Vererbung, access, ...)

```
namespace Humboldt_Universität {
  class Fachbereich { /*...*/ };
  class Student;
  void register(Fachbereich&,
                Student&)
} // ; muss hier nicht stehen im Gegensatz zu class !
```



Namespaces in C++

- *namespace reopening*: zusätzliche Deklaration, fehlende Definitionen

```
namespace Humboldt_Universität {
  void register(Fachbereich& f, Student& s)
  {
  //...
  }
  // ...
} // gehört zum gleichen namespace !
```

Namespaces in C++

- Aliases: verkürzte Formen

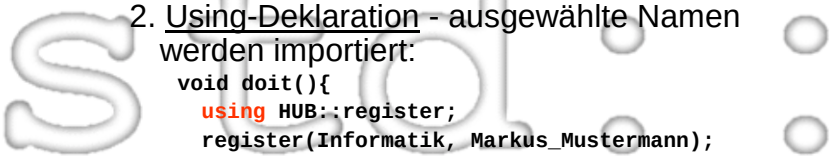
```
namespace HUB = Humboldt_Universität;
```

- zwei Möglichkeiten der "Bereitstellung" von Elementen aus namespaces:
  1. Using-Direktive - alle Namen werden importiert:

```
using namespace Humboldt_Universität;
Fachbereich Informatik;
Student Markus_Mustermann;
```

  2. Using-Deklaration - ausgewählte Namen werden importiert:

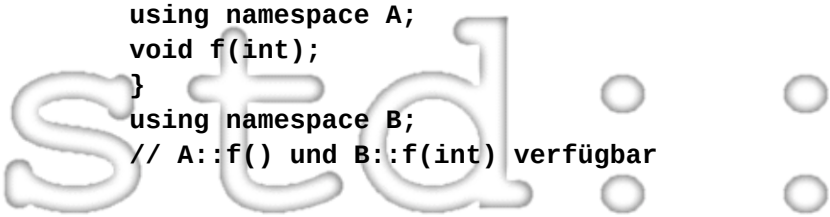
```
void doit(){
  using HUB::register;
  register(Informatik, Markus_Mustermann);
}
```



Namespaces in C++

- *Lookup* unqualifizierter Namen: verwende alle using-Deklarationen, und rekursiv alle using-Direktiven:

```
namespace A{
void f();
}
namespace B{
using namespace A;
void f(int);
}
using namespace B;
// A::f() und B::f(int) verfügbar
```



Namespaces in C++

- *Lookup* qualifzierter Namen: Alle using-Deklaration; using-Direktiven nur, wenn zuvor nichts gefunden wurde

```
namespace A{
  void f(int);
}
using namespace A;
using B::f;
void g1(){
   ::f(); //B::f
}
void f(int);
void g2() {
   ::f(3); // OK
   f(3);   // FEHLER
}
```

```
namespace B{
  void f();
}
```


Namespaces in C++

- Koenig-Lookup

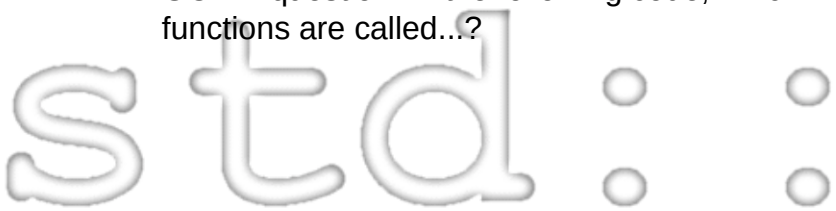
```
namespace Humboldt_Universität{
 ostream& operator <<(ostream &, Student&);
 Student* suche(Fachbereich&, char*); }

void f(char *name)
{
  HUB::Student* s = suche (name);
  cout << "Found" << *s << endl;
  // geht nicht mit den bisherigen lookup-Regeln !
}
void f(char *name) // äußerst unhandlich
{
  HUB::Student* s = HUB::suche (name);
  HUB::operator<<(cout<<"Found",*s)<<endl;
}
```

Namespaces in C++

- Koenig-Lookup (simplified):

  If you supply a function argument of class type, then to look up the correct function name the compiler considers matching names in the namespace containing the arguments type

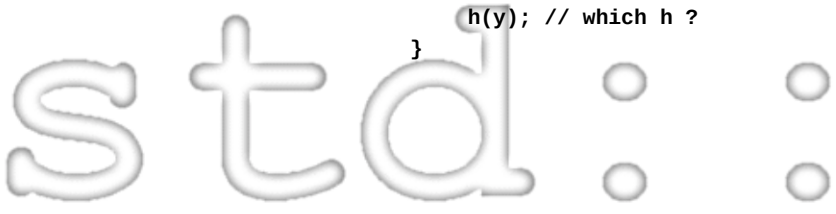
- Herb Sutter: Exceptional C++, Item 31:
  - GOTW question: In the following code, which functions are called...?



Namespaces in C++

```
namespace A {
   struct X;
   struct Y;
   void f(int);
   void g(X);
}
```

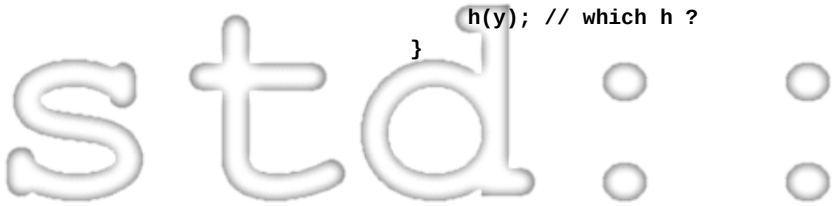
```
namespace B {
   void f(int i) {
        f(i); // which f ?
   }

   void g (A::X x){
        g(x); // which g ?
   }

   void h (A::Y y){
        h(y); // which h ?
   }
```



Namespaces in C++

```
namespace A {
   struct X;
   struct Y;
   void f(int);
   void g(X);
}
```

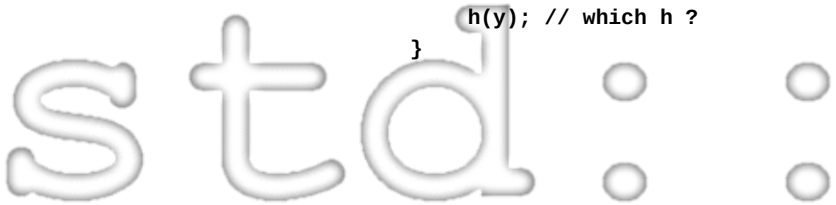
```
namespace B {
   void f(int i) {
        f(i); // B::f !
   }

   void g (A::X x){
        g(x); // which g ?
   }

   void h (A::Y y){
       h(y); // which h ?
   }
```



Namespaces in C++

```
namespace A {
   struct X;
   struct Y;
   void f(int);
   void g(X);
}
```

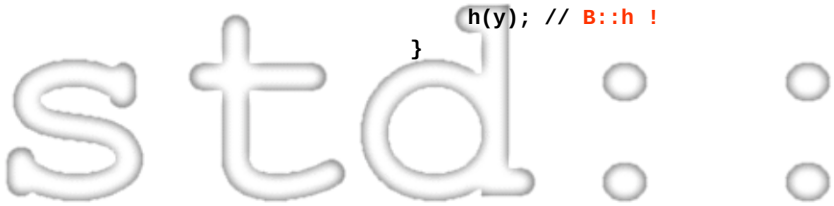
```
namespace B {
   void f(int i) {
        f(i); // B::f !
   }

   void g (A::X x){
        g(x); // AMBIGUOUS !!!
   }

   void h (A::Y y){
       h(y); // which h ?
   }
```



Namespaces in C++

```
namespace A {
   struct X;
   struct Y;
   void f(int);
   void g(X);
}
```

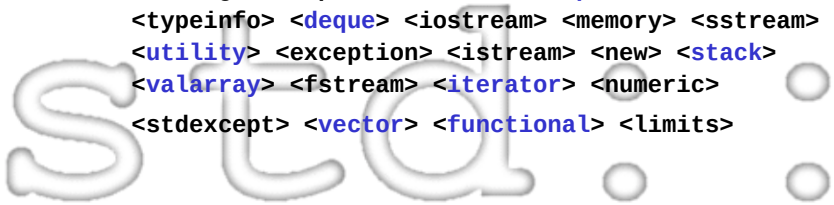
```
namespace B {
   void f(int i) {
        f(i); // B::f !
   }

   void g (A::X x){
        g(x); // AMBIGUOUS !!!
   }

   void h (A::Y y){
        h(y); // B::h !
   }
```



namespace std

- **Alle Symbole der Bibliothek** (außer Makros, `operator new` und `operator delete`) **sind im Namensraum std** (oder in lokalen Namensräumen von std) **definiert.**

```
<algorithm> <iomanip> <list> <ostream>
<streambuf> <bitset> <ios> <locale> <queue>
<string> <complex> <iosfwd> <map> <set>
<typeinfo> <deque> <iostream> <memory> <sstream>
<utility> <exception> <istream> <new> <stack>
<valarray> <fstream> <iterator> <numeric>
<stdexcept> <vector> <functional> <limits>
```



namespace std

- **Symbole der Standard C Bibliothek werden durch zusätzliche Headerfiles bereitgestellt (ebenfalls lokal zu std !):**

```
<cassert> <ciso646> <csetjmp> <cstdio> <ctime>
<cctype> <climits> <csignal> <cstdlib>
<cwchar> <cerrno> <clocale> <cstdarg>
<cstring> <cwctype> <cfloat> <cmath> <cstddef>
```


namespace std

ALT UND NICHT STANDARDKONFORM

```
#include <iostream.h>
#include <string.h>
int main()
{ char s[20];
  strcpy (s, "Hello, world!");
  cout << s << endl;
}
```

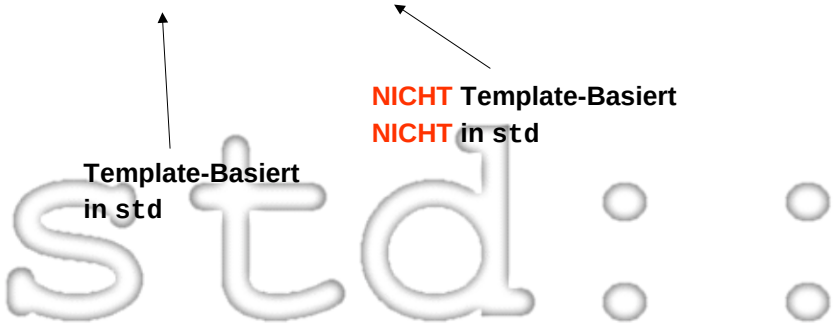
STANDARDKONFORM

```
#include <iostream>
#include <cstring>
int main()
{ char s[20];
  std::strcpy(s, "Hello, world!");
  std::cout << s << std::endl;
}
```

namespace std

ACHTUNG: einige C++ -Implementationen (z.B. VC++6) bieten z.T. noch Headerfiles nach alten und neuen Regeln an:

`<iostream>` und `<iostream.h>`



Exceptions

- **Behandlung von Fehlerzuständen zur Programmlaufzeit: traditionelle Ansätze (returning Error-Codes, errno, ...) sind unzureichend**
- **Fehlerursache und Fehlerbehandlung liegen in verschiedenen Verantwortlichkeitsbereichen:**

```
// Modul: any_lib.cc:
int f (int i,int j){
  // never call with j==0
  return i /j; // hier tritt der Fehler evtl. auf
}
```

```
// Modul: Benutzung:
#include "any_lib.h"
main() {
  int n =f (1,0);
  // hier ist der Fehler eigentlich zu behandeln
}
```

Exceptions

- **C++ (wie Ada, Java):**
  Konzept des *exception handling*:
  - **eine Folge von Anweisungen kann in einen sog. `try`-Block eingeschlossen werden, an den sich eine Folge von `catch`-Blöcken anschließen kann,**
  - **Sobald während der Ausführung des `try` -Blockes ein Fehler "aufgeworfen" wird (u.U. aus indirekt gerufenen Funktionen), wird die Abarbeitung des `try` –Blockes beendet und in einen passenden `catch`-Block verzweigt !**
  - **Ggf. werden `catch`-Blöcke entlang der Aufrufkette abgesucht**
  - **Beim Auftreten vom Exceptions werden für alle lokalen Objekte (die erfolgreich konstruiert wurden) die Destruktoren vor Verlassen des `try`-Blockes gerufen !**

Exceptions

```
// Modul: any_lib.cc:
int f (int i,int j) {
   if (!j) throw anException;
   return i /j;
}
```

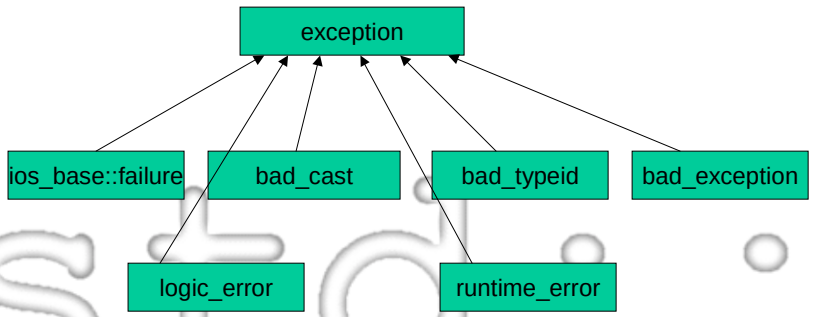
```
// Modul: Benutzung:
#include "any_lib.h"
main() {
  try {
        int n =f (1,0);
  }
  catch (anException) {
        std::cerr << "Division durch Null\n„
  }
}
```

Exceptions

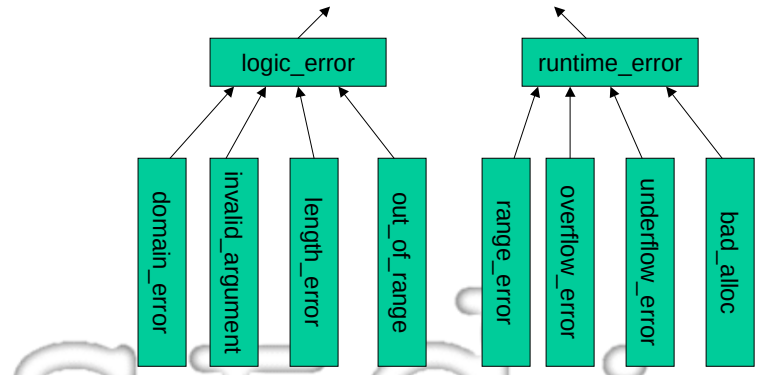
- **Was ist eine Exception ?**
  Es ist ein Objekt eines beliebigen Typs, welches vom Ort des Auftretens des Fehlers (als Kopie) an den Behandlungsblock übergeben wird !
- **ein *Handler* vom Typ `T`, `const T`, `T&` oder `const T&` passt zu einem `throw`-Ausdruck mit einem Objekt des Typs `E` wenn**
  - **[1] `T` und `E` der gleiche Typ sind, oder**
  - **[2] `T` eine `public` Basis von `E` ist, oder**
  - **[3] `T` und `E` Zeigertypen sind, und `E` kann durch Anwendung der Standardtypkonversion in `T` umgewandelt werden**
- **Exceptions können Hierarchien bilden und die Fehlerbehandlung kann vom Speziellen zum Allgemeinen hin erfolgen**

Exceptions

- **Der Typ einer Exception wird zur LAUFZEIT ausgewertet: Implemenation setzt RTTI voraus!**
- **Vordefinierte Exceptions `<exception>`:**



Exceptions



Exceptions

Nicolai Josuttis: ***„Die C++ Standardbibliothek ist ein sehr heterogenes Gebilde. Aus verschiedenen Quellen wurde eine einheitliche Bibliothek ‚zusammengestückelt' und dann in relativ kurzer Zeit angeglichen. Dabei ist allerdings kein völlig homogenes Gebilde entstanden, was sich auch in der Fehlerbehandlung widerspiegelt. So gibt es Teile ..., die eine sehr ausführliche Fehlerprüfung durchführen ... . Andere Teile, wie die STL gehen davon aus, dass der Programmierer weiß, was er tut, und prüfen so gut wie gar nichts." [NJ S. 21]***

Exceptions

***Exception Safety*: Code, der auch beim Auftreten von Exceptions korrekt/definiert arbeitet (Programmzustand ist definiert)**

***Exception Neutrality*: Code, der alle auftretenden Exceptions an den Aufrufer weiterleitet**

Auswirkungen von Exceptions sind schwerwiegend:

1. **Coding Style :**

```
try { X* x = new X;
      foo(x); // potential leak here
      delete x;
}
```


Exceptions

Lösung: "Resource Acquisition is Initialization"

```
class Xpointer{ X* _p;
public:
  Xpointer(X* p):_p(p){}
  ~Xpointer(){ delete _p; }
  operator X* () { return _p; }
};
try {    Xpointer x(new X);
         foo(x); // no leak possible !
}
```

Exceptions

2. Code Review :

(Herb Sutter: Exc. C++, Item 18)

Wie viele Ausführungspfade sind im folgenden Code-Fragment möglich:

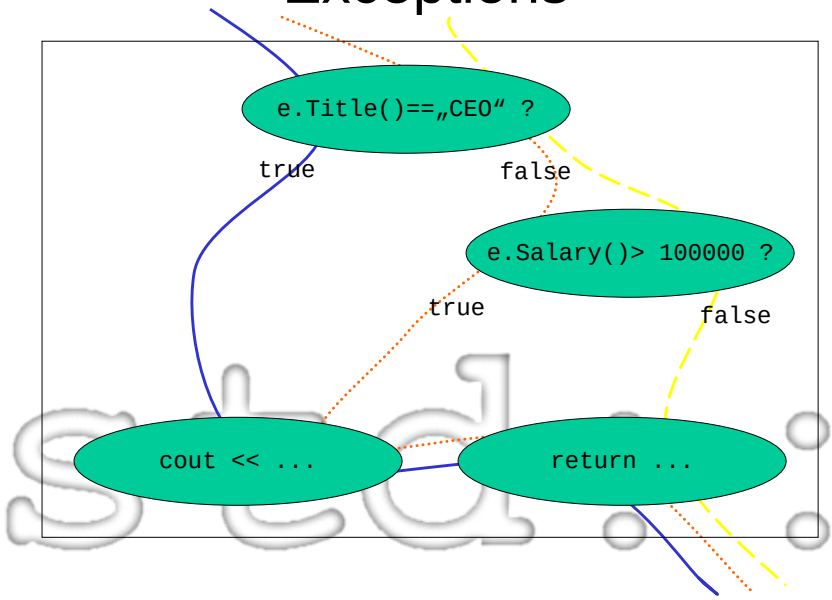
```
String EvaluateSalaryAndReturnName (Employee e)
{
  if (e.Title() == "CEO" || e.Salary() >100000)
  {
    cout<<e.First() << " "<<e.Last()<< " is overpaid" <<endl;
  }
  return e.First() + " " + e.Last();
}
```


Exceptions

If you found: 3

Rate yourself: Average

Exceptions



Exceptions

If you found:	Rate yourself:
3	Average
4 - 14	Exception-aware

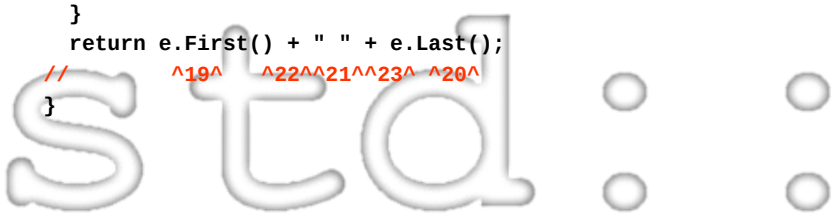
Exceptions

If you found:	Rate yourself:
3	Average
4-14	Exception-aware
15-23	Guru material

3 Pfade ohne Exceptions
20 Pfade mit Exceptions !!!!

Exceptions

```
String EvaluateSalaryAndReturnName (Employee e)
//^*^                                        ^4^
{
  if (e.Title() == "CEO" || e.Salary() >  100000)
//     ^5^     ^7^  ^6^ ^11^   ^8^    ^10^  ^9^
  {
    cout<<e.First() << " "<<e.Last()<< " is overpaid" <<endl
//     ^12^ ^17^   ^13^  ^14^ ^18^ ^15^              ^16^
  }
  return e.First() + " " + e.Last();
//       ^19^   ^22^^21^^23^ ^20^
}
```



Exceptions

Exception Safety und die Standard Bibliothek (Container und Iterator part – STL):

H. Sutter ([HS1 S.59]): Sind die Container der Standard Bibliothek exception-safe (ES) und exception-neutral?

Die kurze Antwort ist: Ja.

ABER ... Kopie-Konstruktoren der Elemente könnten scheitern – copy -> change -> swap

`vector<T>` und `deque<T>` nur ES wenn T-Kopie und T-Zuweisung keine Exceptions werfen

Templates

Parametisierte (generische) Typen und Funktionen:

- **Klassen- bzw. Funktionsdefinitionen, die noch von weiteren Typen oder Werten abhängen:**

```
#define MAX(X,Y)  (X) > (Y) ? (X) : (Y)

inline int max (int i, int j)
{ return i > j ? i : j; }

template <class T>
inline const T& max (const T& i, const T& j)
{ return i > j ? i : j; }
```

Templates

Parametisierte (generische) Typen und Funktionen:

```
template <class T, int S>
class Stack {
   T elements[S]; ....
};
```

Erzeugung der konkreten Typen zur Übersetzungszeit == INSTANZIERUNG

```
Stack<int>  Stack< Stack<char> >
max(3.0, 5.4) max<int>(3.0, 5.4)
```

Templates

Der Compiler kann bei der Übersetzung rekursive Berechnungen anstellen!

Für gewisse Template-Parameter können spezialisierte (vom allg. Muster abweichende) Implementierungen angegeben werden!

```
template <int N>
class Fib {
       long val;
public:
       Fib():val(Fib<N-1>()+Fib<N-2>()){}
       operator long () {return val;}
};
```

Templates

```
template<>
class Fib<0> {
        long val;
public:
        Fib():val(1){}
        operator long () {return val;}
};

template<>
class Fib<1> {
        long val;
public:
        Fib():val(1){}
        operator long () {return val;}
};
```


Templates

Ein Template kann beliebige Eigenschaften von Parametertypen verwenden, erst bei der konkreten Instanzierung wird entschieden, ob korrekter Code entsteht (nur der wirklich gerufene Code muss korrekt sein!!)

Dadurch ist u.U. nicht mehr klar, ob ein Bezeichner der Parameterklasse ein Typ oder etwas anderes benennt:

```
template <class T> class X {
        void foo() { T::K * p; ... // ?????}
}; // Multiplikation oder Zeigervereinbarung ?
```

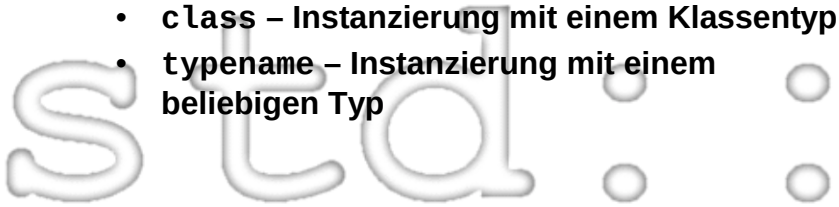
Templates

```
template <class T> class X {
      void foo() { typename T::K * p; ...}
};
```

typename kann auch anstelle von `class` als Template-Parameter benutzt werden:

Übliche Konvention (Empfehlung):

- **`class` – Instanzierung mit einem Klassentyp**
- **`typename` – Instanzierung mit einem beliebigen Typ**



Templates

Templates können Default-Argumente haben:

```
template <class T, class C = vector<T> >
class Sample;

Sample<int>  ---- Sample<int,vector<int> >
```

Templates

Memberfunktionen von (Template-)Klassen können (unabhängige) Template-Funktionen sein:

```
// Member-Templates
class X { ...
     template <class P> void foo(P);
};
Template <class Q>
class Y { ...
     template <class P> void foo(P);
};
```


Templates

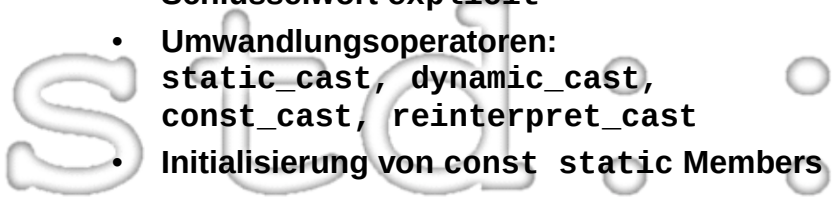
Template-Parameter können selbst Template-Klassen sein (wird in `std` nicht benutzt !):

```
// class template SmartPtr (definition) LOKI, A. Alexandrescu
template
<   typename T,
    template <class> class OwnershipPolicy,
    class ConversionPolicy,
    template <class> class CheckingPolicy,
    template <class> class StoragePolicy
>
class SmartPtr: public StoragePolicy<T>,
public OwnershipPolicy<typename StoragePolicy<T>::PointerType>,
public CheckingPolicy<typename StoragePolicy<T>::StoredType>,
public ConversionPolicy
    { ..... };
```

Templates

Außerdem muss ein `std`-konformer Compiler beherrschen:

- **Typ `bool`**
- **lokale Template-Klassen**
- **Exception Handling**
- **Namespaces**
- **Schlüsselwort `explicit`**
- **Umwandlungsoperatoren: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`**
- **Initialisierung von `const static` Members**



Templates

Templates erlauben eine neue Form von Polymorphie (Vielgestaltigkeit):

Beispiel: H. Sutter, More Exc. C++, Item 1

What is the best way to dynamically use different stream sources and targets, including the standard console streams and files?

```
echo infile outfile
echo <infile >outfile
```

BTW: Welche Typen haben `std::cin` und `std::cout` ?

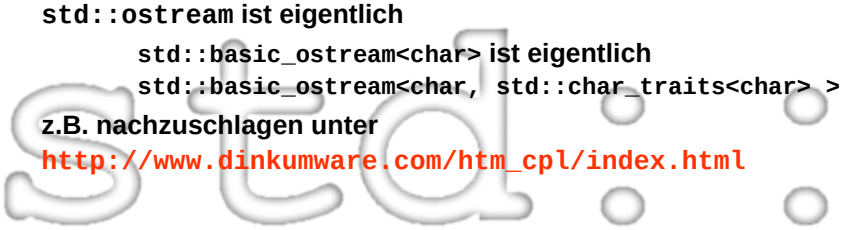
Templates

`std::cin` hat den Typ `std::istream` und
`std::cout` hat den Typ `std::ostream`

`std::istream` ist eigentlich
`std::basic_istream<char>` ist eigentlich
`std::basic_istream<char, std::char_traits<char> >`

`std::ostream` ist eigentlich
`std::basic_ostream<char>` ist eigentlich
`std::basic_ostream<char, std::char_traits<char> >`

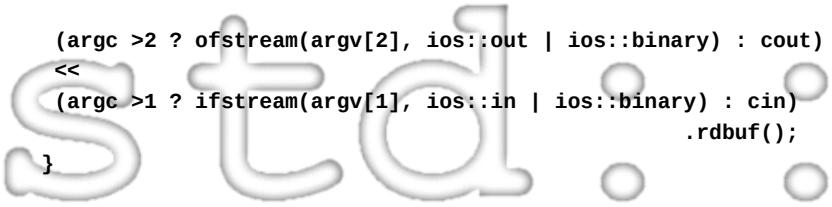
z.B. nachzuschlagen unter
http://www.dinkumware.com/htm_cpl/index.html



Templates

The tersest solution (by H. Sutter):

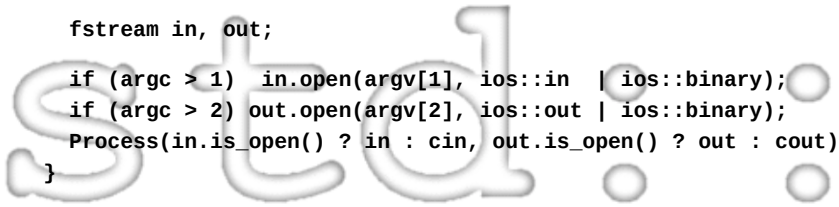
```
// Example 1-1: A one-statement wonder

#include <fstream>
#include <iostream>

int main (int argc, char* argv[])
{
 using namespace std;

 (argc >2 ? ofstream(argv[2], ios::out | ios::binary) : cout)
 <<
 (argc >1 ? ifstream(argv[1], ios::in | ios::binary) : cin)
                                                  .rdbuf();
}
```



Templates

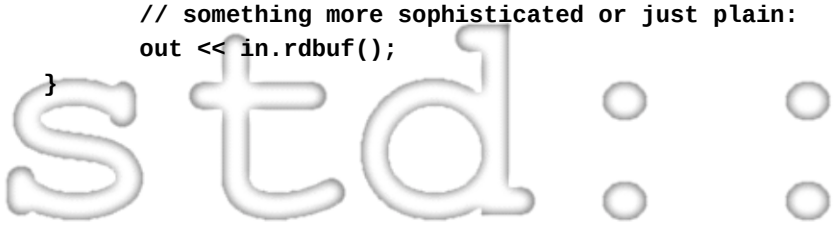
A better solution:

```
#include <fstream>
#include <iostream>

void Process (???, ???);

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);
}
```



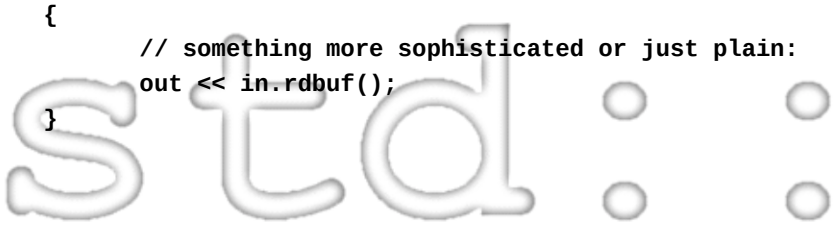
Templates

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



Templates

```
// Run-Time Polymorphism (better):
template <typename C=char, typename T=char_traits<C> >
void Process
(
      std::basic_istream<C, T>& in,
      std::basic_ostream<C, T>& out
)
{
      // something more sophisticated or just plain:
      out << in.rdbuf();
}
```



Templates

Warum dann nicht gleich:

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

MAXIMALE ENTKOPPLUNG ! *(Prefer Extensibility)*

Templates

Helpers from <utility>:

```
namespace std {
  namespace rel_ops { // warum denn das ???
    template <class T>
    inline bool operator!=(const T& x, const T& y)
    {
      return !(x == y);
    }

    template <class T>
    inline bool operator>(const T& x, const T& y)
    {
      return y < x;
    } ....// <=, >=
  } /* End of namespace rel_ops */
```

Templates

```
// Pairs.
//    immer noch in std...

  template <class T1, class T2>
  struct pair
  {
    typedef T1 first_type;  // in allen Containern
    typedef T2 second_type; // anzutreffen

    T1 first;
    T2 second;
    pair (const T1& a, const T2& b)
            : first(a), second(b) {}
    pair (): first(T1()), second(T2()) { }

    pair(const pair& p)
      : first(p.first), second(p.second) { }
```

Templates

```
// weiter in pair
//    immer noch in std...

   template <class U, class V> pair(const pair<U,V>& p)
      : first(p.first), second(p.second) {  }
  }; // end of pair

  template <class T1, class T2>
  inline bool operator==
       (const pair<T1, T2>& x, const pair<T1, T2>& y)
  {
    return x.first == y.first && x.second == y.second;
  }  // <, !=, >, >=, <=
```


Templates

```
// and now we proudly present:

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y)
  {
    return pair<T1, T2>(x, y);
  }

} // end of this std section
```