static_cast<T>

» den Compiler überreden, verwandte Typen verträglich zu verwenden «
das Ergebnis kann ohne erneute Umwandlung verwendet werden

```
class X { ... };
class Y : public X {};

// eine Y& ist auch immer eine X&
Y o;
X& x1 = o; // implizite Anpassung der Typen
X& x2 = static_cast<X&> (o); // dasselbe

// manchmal ist eine X& auch eine Y&
Y& y1 = static_cast<Y&> (x1); // ok, weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X; Y& y2 = static_cast<Y&> (x3); // Crash ahead
```

reinterpret_cast<T>

» den Compiler überreden, nicht verwandte Typen verträglich zu verwenden «

das Ergebnis kann nur nach erneuter Rückumwandlung verwendet werden

die unveränderte Bitbelegung wird anders interpretiert; zumeist nicht portabel

```
int *pi = &someint;
void *v = reinterpret_cast<void*>(pi);
// don't use v, but:
int *p = reinterpret_cast<int*>(v);
*p = 337; // OK: sets someint
```

dynamic_cast<T>

» zur Laufzeit verwandte Typen verträglich und sicher verwenden «

```
class X { virtual void foo(); };
class Y : public X {};


// manchmal ist eine X& auch eine Y&
Y& y1 = dynamic_cast<Y&> (x1); //ok,weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X;
Y& y2 = dynamic_cast<Y&> (x3); // NO Crash ahead
// exception std::bad_cast !!!
```

dynamic_cast<T>

Implementation setzt offenbar Auswertung von Laufzeittypinformationen (RTTI - run time type identification) voraus

Funktioniert für Zeiger und Referenzen polymorpher Typen (es muss virtuelle Funktionen in der Basisklasse geben!)

Ein downcast (von einem Zeiger/einer Referenz auf eine Basisklasse auf einen Zeiger/eine Referenz einer Ableitung) gelingt, wenn das referenzierte Objekt vom Typ der Ableitung oder einer Ableitung dieser ist.

Bei Zeigern liefert `dynamic_cast` den Wert 0, bei Referenzen wird die Ausnahme `std::bad_cast` geworfen, wenn die dynamische Typ nicht ausreicht

dynamic_cast<T>

```
class A {
public:  virtual void needed () {}
};
class B: public A {public: int i;};
class C: public B {public: int j;};
int main() {
        A *pa = new B;
        B *pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
        C *pc = dynamic_cast<C*>(pb);
        if (pc) pc->j = 54321; // wird nicht ausgefuehrt
        pa = new C;
        pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
}
```

Darüber hinaus kann man die Typidentität direkt abfragen:

dazu existiert der Operator `typeid` (wie `sizeof` vom Compiler umgesetzt und nicht überladbar), der eine (vergleichbare) Struktur des Typs `type_info` liefert, der Vergleich von `type_info` gelingt, wenn exakt der gleiche Typ vorliegt

auf `type_info` ist wiederum die Funktion `name()` definiert, die einen Klarnamen der Klasse (nicht notwendig identisch mit dem Klassennamen) erzeugt

`#include <typeinfo>` ist erforderlich

Die beteiligten Typen müssen wiederum polymorph sein, d.h. mindestens eine virtuelle Funktion in der gemeinsamen Basis besitzen

```
#include <typeinfo>
#include <iostream>
using namespace std;

class A { virtual void any () {} };
class B: public A { };
class C: public A { };
void check (A* p) {
        if (typeid(*p)==typeid(A))
                { cout << "es ist ein A\n"; return; }
        if (typeid(*p)==typeid(B))
                { cout << "es ist ein B\n"; return; }
        cout << "weder A noch B\n";
}
const char* get_name(A* p) {
        return typeid(*p).name();
}
```

```
int main()
{
        A *p;
        p = new A;
        check (p);
        cout << get_name(p) << endl;
        p = new B;
        check (p);
        cout << get_name(p) << endl;
        p = new C;
        check (p);
        cout << get_name(p) << endl;
}
```

```
C:\tmp>rtti
es ist ein A
A
es ist ein B
B
weder A noch B
C
```

dynamic_cast<T> ist manchmal nicht zu vermeiden

```
class B {
        // no functionality 'foo'
};
class D: public B {
        virtual void foo();
};
void register (B*);
B* next();
...
register(new D);
...
B* n = next();

// how to call foo ?
dynamic_cast<D*>(n)->foo();
```

RTTI nur in Ausnahmefällen explizit benutzen

statt spaghetti code

```
Shape * s;
if (typeid(*s)== typeid("Circle"))
((Circle*)s)->Circle::draw();
else
if (typeid(*s)== typeid("Rectangle"))
        ((Rectangle*)s)-> Rectangle::draw();
else ...
```

benutze

```
Shape * s;
s->draw(); // late bound virtual
```

Mehrfachvererbung (multiple inheritance)

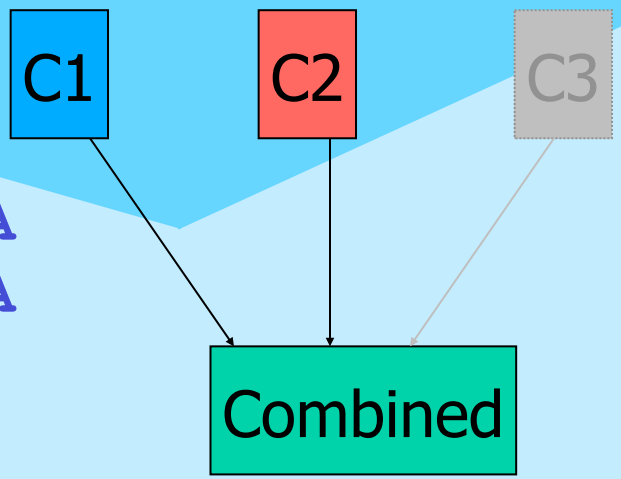
eine Klasse kann mehrere Basisklassen haben --> 'freie' Kombination von Konzepten:

```
class Combined:      public Concept1,
                     public Concept2,
                     private Concept3
{ .... };
```

jedes `Combined`-Objekt IST EIN `Concept1` und IST EIN `Concept2` (`Concept3`-Abstammung ist ein Implementationsdetail)

Mehrfachvererbung (multiple inheritance) Polymorphie bleibt erhalten:

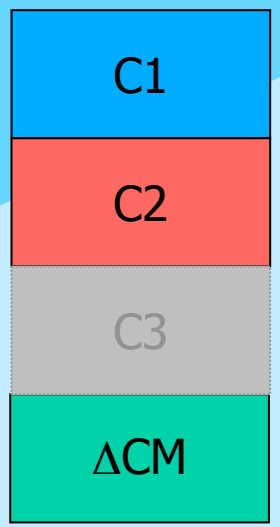
```
Combined obj;
Combined * cm = &obj;
Concept1 * c1 = cm; // IS A
Concept2 * c2 = cm; // IS A
// NOT c2=c1=cm;

// it's the same Object:
if (c1 == cm) // yes !
if (c2 == cm) // yes !
if (c1 == c2) // ERROR: uncomparable
```



Mehrfachvererbung (multiple inheritance)

Layout muss linearisiert werden:

```
Combined obj;
Combined * cm = &obj;
Concept1 * c1 = cm; // IS A
Concept2 * c2 = cm; // IS A
```




```
// but the (numeric) adresses may differ:
if(reinterpret_cast<void*>c1==reinterpret_cast<void*>cm)
        // yes or no!
if(reinterpret_cast<void*>c2==reinterpret_cast<void*>cm)
        // yes or no!
```

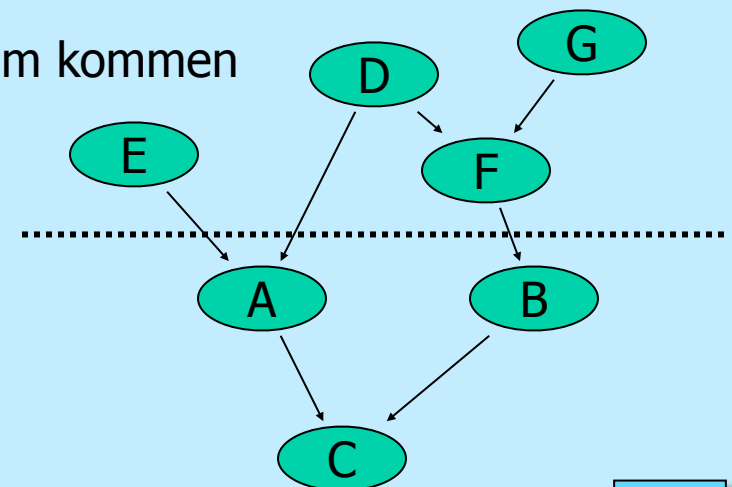
Mehrfachvererbung (multiple inheritance)

eine Klasse kann eine andere nicht direkt mehrfach einerben

```
struct A { int i; };
class B: public A, public A {   // NOT ALLOWED
       void foo(){ i = 0; /* which i ? A::i ? which A ? */ }
};
```

ansonsten kann es durchaus zu Maschen im Baum kommen

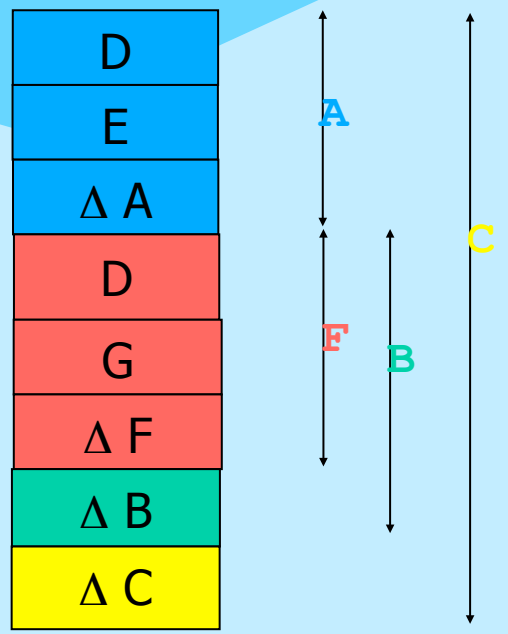
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
-----------------------------------
class C: public A, public B {...};
```



Mehrfachvererbung (multiple inheritance)

ob dabei der mehrfach eingeerbte Basisklassenanteil dupliziert oder unifiziert im resultierenden Objekt erscheint ist steuerbar:

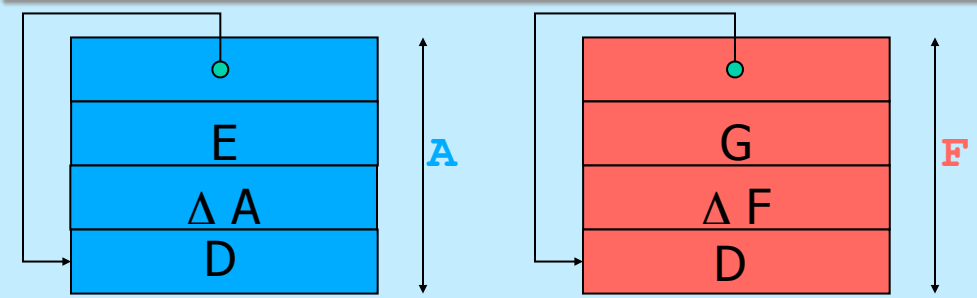
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
class C: public A, public B {...};
```

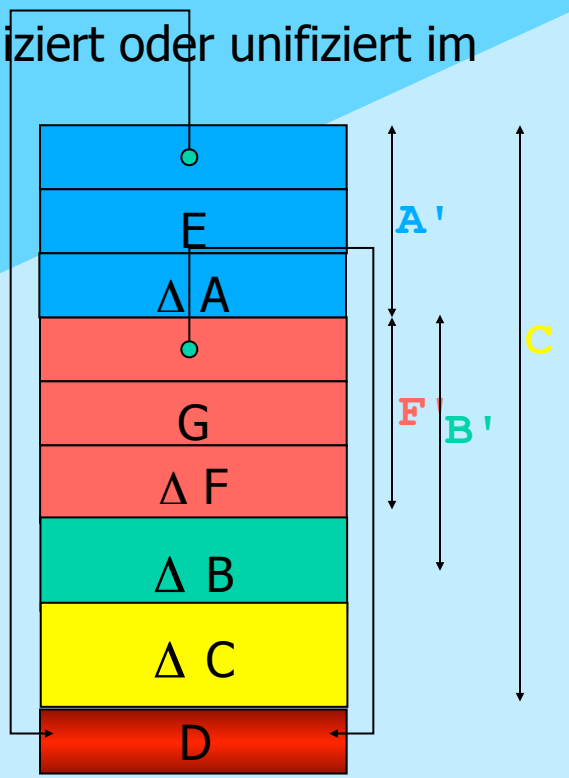


Mehrfachvererbung (multiple inheritance)

ob dabei der mehrfach eingeerbte Basisklassenanteil dupliziert oder unifiziert im resultierenden Objekt erscheint ist steuerbar:

```
class A: virtual public D,
              public E {...};
class F: virtual public D,
              public G {...};
class B: public F {...};
class C: public A, public B {...};
```





Mehrfachvererbung (multiple inheritance)

für beide Szenarien gibt es sinnvolle Anwendungen:

```
// class Listable {/* Listeneigenschaften */ };
class A: public Listable { ... };
// A's können in einer Liste erfasst werden
class B: public Listable { ... };
// B's können in einer Liste erfasst werden
class C: public A, public B { ... };
// C's können in zwei separaten Listen (als A und als B)
// erfasst werden
```

non virtual

---

virtual

```
class Person {...};
class Angestellter: public virtual Person {...};
class Student: public virtual Person {...};
class Werkstudent: public Angestellter, public Student
{...}; // ein und dieselbe Person !!!
```

Mehrfachvererbung (multiple inheritance)

Durch die freie Kombination kann es leicht zu Mehrdeutigkeiten kommen

Falls diese nicht auflösbar sind, liegt ein statischer Fehler vor (s.o. `B: A,A`)
Aber auch:

```
class A {public: int i;};       class B: public A{};

class C: public A, public B {}; // ERROR
// i ... which i ? A::i ? which A::i ?
```

Mehrdeutigkeiten, die durch scope resolution auflösbar sind, sind erlaubt

```
struct A { int i; };  struct B { int i; };
class C: public A, public B {     i=1; // ERROR
                                  A::i=1; // OK
                                  B::i=1; // OK
};
```

Mehrfachvererbung (multiple inheritance)

Selbst bei virtuellen Basisklassen kann es auf Grund der Maschenbildung sein, dass ein Name eines Members auf mehreren "Wegen" auflösbar ist und zu verschiedenen Membern führt, Eindeutigkeit liegt dann vor, wenn es (genau) einen kürzesten Weg gibt »Dominanzregel« (ansonsten muss ebenfalls qualifiziert werden)

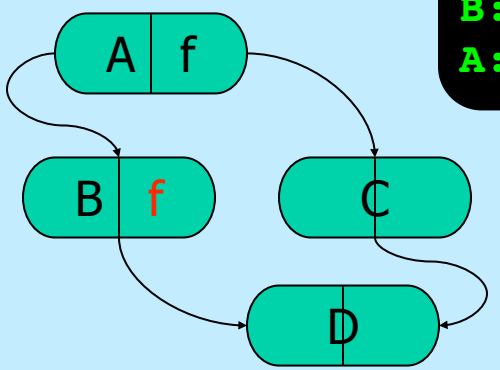
```
class A { public: void f(){cout<<"A::f()\n";}   };
class B: public virtual A {
          public: void f(){cout<<"B::f()\n";}
};
class C: public virtual A {
          public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {};
int main() {   D d;
               // d.f(); ERROR: ambiguous access of 'f'
               d.A::f(); // ok
               B *pb = &d; pb->f(); // ok
               C *pc = &d; pc->f(); // ok
}
```

Mehrfachvererbung (multiple inheritance)

Selbst bei virtuellen Basisklassen kann es auf Grund der Maschenbildung sein, dass ein Name eines Members auf mehreren "Wegen" auflösbar ist und zu verschiedenen Membern führt, Eindeutigkeit liegt dann vor, wenn es (genau) einen kürzesten Weg gibt »Dominanzregel« (ansonsten muss ebenfalls qualifiziert werden)

```
class A { public: void f(){cout<<"A::f()\n";}  };
class B: public virtual A {
           public: void f(){cout<<"B::f()\n";}
};
class C: public virtual A {};
class D: public B, public C {};
int main() {   D d;
               d.f();
               d.A::f();
               B *pb = &d; pb->f();
               C *pc = &d; pc->f();
}
```

```
B::f()
A::f()
B::f()
A::f()
```

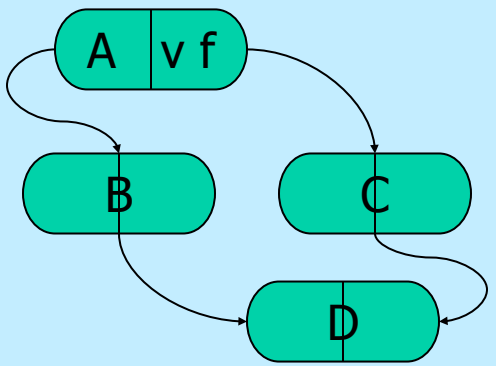


Mehrfachvererbung (multiple inheritance)

Mehrfachvererbung und virtuelle Funktionen sind miteinander kombinierbar, im Falle von virtuellen Basisklassen stehen u.U. ebenfalls mehrere Wege der Auflösung zur Verfügung: falls keine dominate Implementation existiert, muss in der am weitesten abgeleiteten Klasse eine Redefinition erfolgen

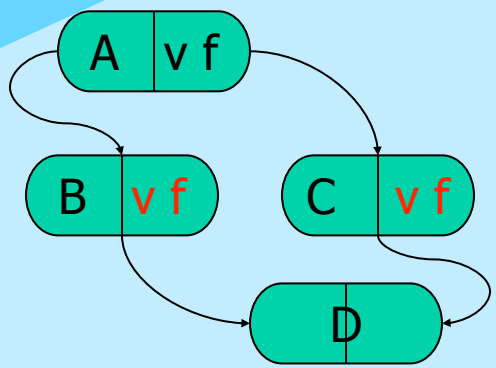
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A { };
class C: public virtual A { };
class D: public B, public C { };
main(){         D d;
                C *pc = &d;
                pc->f();
}
```

```
A::f()
```



Mehrfachvererbung (multiple inheritance)

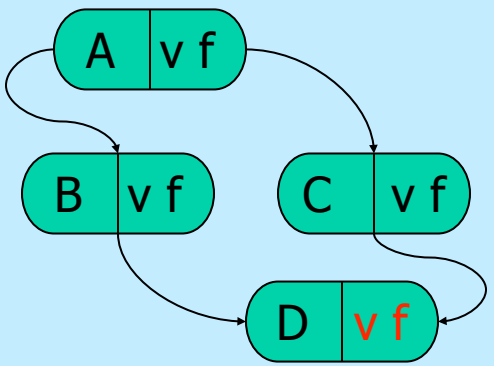
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C { };
// ERROR: no unique final overrider for f() in D
```



Mehrfachvererbung (multiple inheritance)

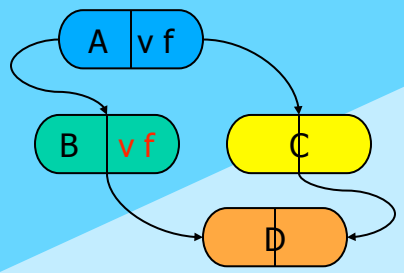
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {
  public: void f(){cout<<"D::f()\n";}
};
... D d; C *pc = &d; pc->f(); ...
```

```
D::f()
```

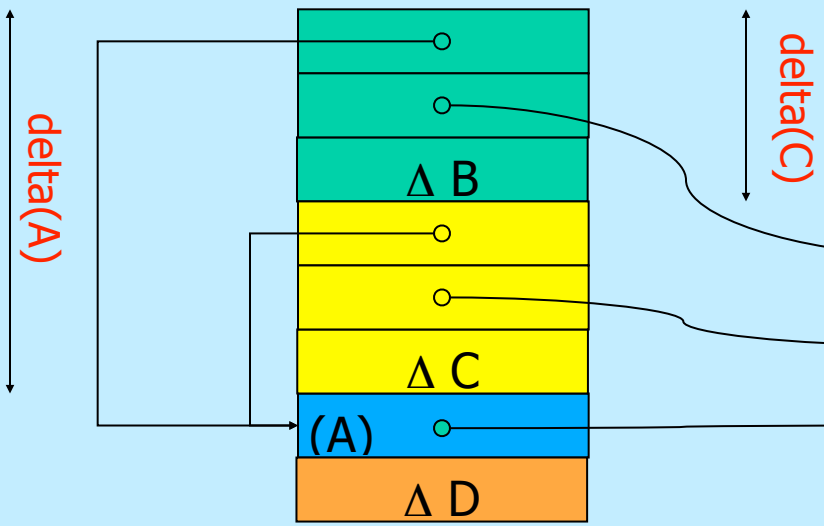


Mehrfachvererbung (multiple inheritance)

Implementation von virtuellen Funktionen wird 'slightly more complicated'




```
main(){
        D d;
        C *pc = &d;
        pc->f();
}
```




```
struct vtbl_entry {
  void (*fct)();
  int delta;
};
```

B::f	0
B::f	- delta(C)
B::f	- delta(A)

Mehrfachvererbung (multiple inheritance)

Implementation von virtuellen Funktionen wird 'slightly more complicated'

```
typedef struct vtbl_entry {
  void (*fct)();
  int delta;
} VE;
```

```
D d;
A* pa = &d; B* pb = &d; C* pc = &d;
pa->f();
// VE* vt = &pa->vtbl[index(f)];
// (*vt->fct)((B*)((void*)pa + vt->delta));
pb->f();
// VE* vt = &pb->vtbl[index(f)];
// (*vt->fct)((B*)((void*)pb + vt->delta));
pc->f();
// VE* vt = &pc->vtbl[index(f)];
// (*vt->fct)((B*)((void*)pc + vt->delta));
```

Mehrfachvererbung (multiple inheritance)

Konstruktoren virtueller Basisklassen müssen in der am weitesten abgeleiteten Klasse direkt gerufen werden !

```
class A { public: A(int); };
class B: public virtual A {
  public: B(): A(1){ .... }
};
class C: public virtual A {
  public: C(): A(2){ .... }
};

class D: public B, public C {
// public: D() { .... } // ERROR: no matching function for call to `A::A ()'
   public: D(): A(3) { .... }
};
```

Mehrfachvererbung (multiple inheritance)

Potentielle Mehrdeutigkeiten werden unabhängig von Zugriffsrechten lokalisiert !

```
class A {
  private: void m();
};
class B {
  public:  void m();
};
class C: public A, public B {
   void f() {
     // m(); // Fehler: Mehrdeutigkeit
     A::m(); // Fehler: kein Zugriff
     B::m(); // ok
   }
};
```

Mehrfachvererbung (multiple inheritance)

Wird eine virtuelle Basisklasse sowohl `private` als auch `public` vererbt, so dominiert `public` ! Bei nicht virtueller Vererbung gilt für jedes Auftreten einer Basisklasse das Zugriffsrecht entsprechend der direkten Vererbung

```
class A { public: int i; };
```

```
class B: private virtual A {};
class C:  public virtual A {};
class D: public B, public C {
  void f() { i++; /* erlaubt, da B: .... public A */ }
};
-----------------------------------------------------
class B: private A {};
class C: public A {};
class D: public B, public C {
void f() {
  // i++; // Fehler: Mehrdeutigkeit
  C::i++; // ok
  // B::i++; // Fehler: kein Zugriff
};
```

Namespaces

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

Lösung `namespace`: Deklaration wie Klassen, Verschachtelung erlaubt (aber keine Vererbung, Zugriffsrechte, ...)

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

Namespaces dürfen beliebige Deklarationen und Definitionen enthalten (auch Namespaces), Klassen dürfen lokale Klassen enthalten aber keine Namespaces, Typen (Klassen) dürfen nach ihrer Verwendung nicht lokal neu definiert werden

```
namespace X {
        namespace Y {
                typedef int B;
                class A {
                        B i;
// ERROR:                 class B {};     // changes meaning of 'B' from
                                           // 'typedef int X::Y::B'
                        class C {};
                public:
                        class D {};
                };
        }
}
// ERROR:       X::Y::A::C c; // 'X::Y::A::C' is not accessible
                X::Y::A::D d; // OK
```

namespace reopening erlaubt zusätzliche Deklarationen, fehlende Definitionen, logische Verteilung über separate Dateien (nicht für `namespace std` erlaubt)

```
namespace Humboldt_Universitaet { // ...
       void registriere (Fachbereich& f, Student& s)
       {
         // how this is done ...
       }
} // gehört zum gleichen namespace
```

Definitionen auch im umhüllenden namespace möglich

```
class Humboldt_Universitaet::Student {
   //...
};
```

Namen von äußeren namespaces sind wiederum globale Gebilde

--> spricht für lange (und damit) eindeutige Namen

praktische Verwendung

--> spricht für kurze Namen

Lösung: `namespace` Aliasnamen

```
namespace HU = Humboldt_Universitaet;
// as I'll refer it further
```

Es gibt zwei Möglichkeiten der "Bereitstellung" von Elementen aus **namespaces**

1. Mit einer **using**- Deklaration wird ein Name aus einem Namensbereich direkt in den Geltungsbereich eingeführt, in dem die **using** - Deklaration erfolgt (als wäre es dort deklariert worden).

```
void doit(){
   using HU::registriere;
   registriere(Informatik, Markus_Mustermann);
}
```

2. Durch eine `using`- Direktive können sämtliche Namen des angegebenen Namensbereichs für den Geltungsbereich zugreifbar gemacht werden, in dem die `using` - Direktive enthalten ist. Die `using` -Direktive wirkt sich dabei so aus, als seien alle Elemente außerhalb ihres Namensbereichs deklariert, und zwar an der Stelle, an der die Namensbereich-Definition tatsächlich steht.

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

Achtung

`using namespace N;` und `using N::name` (∀ name in N)

sind nicht äquivalent:

```
namespace X {
      int i;
      double x;
}
int main() {
      int i = 1;
      X::i = 10;
      // using X::i;
      // redefinition !!
      using X::x;
      i = 42;
      return 0;
}
```

```
namespace X {
      int i;
      double x;
}

int main() {
      int i = 1;
      X::i = 10;
      using namespace X;
      i = 42;
      return 0;
}
```

Achtung

`using N::anEnumType;` stellt nicht die `enum`-Literale bereit

`using`- Direktiven sollten nie in unbekannten Kontexten (d.h. in denen nicht klar ist, welche Symbole definiert sind) verwendet werden, weil sie dazu führen können, dass Mehrdeutigkeiten oder Verhaltensänderungen entstehen können (Overloading)

--> Vorsicht bei ihrer Verwendung, Benutzung in Header-Files ist "untragbar schlechtes Design" (Josuttis) !!

`using`- Deklarationen können auch benutzt werden, um Zugriff auf Basis-Member abweichend von den sonst geltenden Regeln zu erlauben:

```
class A {
private:    int a1;
protected:  int a2;
            void f(char){}
public:     void f(int){}
            int a3;
};
class B: private A {
public:
  using A::a2;
  using A::f; // all f's
};
```

```
int main() {
        A a;
        B b;
// erlaubt ist:
        a.a3 = 3;
        a.f(0);
        b.a2 = 2;
        b.f('A');
        b.f(1);
}
```

alles was sichtbar ist kann per `using` -Deklaration 'weitergereicht' werden

bei überladenen Funktion müssen alle Varianten zugreifbar sein, sonst liegt ein statischer Fehler vor

Anonyme Namensräume als Ersatz für (static) Objekte mit file scope.

```
namespace {
   int counter = 0;
   void inc();
}
int main(){inc();}

namespace{
   void inc() { counter++;}
}
```

```
namespace { /*body*/ }
                ==
namespace uniqueForThisFile{}
using namespace uniqueForThisFile;
namespace uniqueForThisFile{/*body*/}
```

Lookup unqualifizierter Namen: zunächst lokal (incl. `using`-Deklarationen) und sonst in allen sichtbaren Namespaces (gleichberechtigt)

Namensauflösung erfolgt **immer** in der Reihenfolge
1. lookup
2. overload resolution
3. access check

```
namespace A {
        void f(){cout<<"A::f()\n";}
        void g(){cout<<"A::g()\n";}
}

namespace B {
        void f(char*){cout<<"B::f(char*)\n";}
}

namespace C {
        using namespace A;
        void f(int){cout<<"C::f(int)\n";}
}
```

```
void f(double){cout<<"::f(double)\n";}

int main()
{
      using namespace B;
      using namespace C;

      f(1);
      f(1.0);
      f();
      g();
      f("Hoho");
}
```

```
C::f(int)
::f(double)
A::f()
A::g()
B::f(char*)
```

```
// wie zuvor

int main(){
        using B::f;
        using namespace C;

//      f(1);           // ERROR: passing `int' to argument
                        // 1 of `B::f(char *)' lacks a cast
//      f(1.0);         // ERROR: argument passing to `char *'
                        // from `double'

//      f();            // ERROR: too few arguments to
                        // function `void B::f(char *)'
        g();            // OK
        f("Hoho");      // OK
}
```

Lookup qualifizierter Namen: im jeweils benannten Scope beginnend rekursiv(* in weiteren bis der Name gefunden wird

(* wird bei der Bildung der transitiven Hülle dasselbe Objekt mehrmals gefunden, liegt KEIN Fehler vor

```
namespace A {
        void f(){cout<<"A::f()\n";}
        void g(){cout<<"A::g()\n";}
}

namespace B {
        void f(char*){cout<<"B::f(char*)\n";}
}

namespace C {
        using namespace A;
        void f(int){cout<<"C::f(int)\n";}
}
```

unverändert !

```
// wie zuvor

int main()
{
//      using namespace B;
//      using namespace C;

        C::f(1);
        ::f(1.0);
        A::f();
//      C::f();         // ERROR:  Too few parameters in
                        // call to 'C::f(int)'
        C::g();
        B::f("Blah");
}
```

ändert gar nichts!

Lookup unqualifizierter Namen hat noch einen wichtigen Sonderfall:

Koenig-Lookup alias ADL (argument dependent lookup)

```
namespace N {
  class T {
  public:
    void foo() { N::foo(*this); }
    friend std::ostream& operator<<(std::ostream&, const T&);
    friend void foo (const T&);
  };
}

std::ostream& N::operator<<(std::ostream& o, const T&) {
        return o<<"T-Object"<<std::endl;
}

void N::foo(const T& t) {  std::cout << t; }
```

Koenig-Lookup alias ADL (argument dependent lookup)

aus allen Parametertypen eines Funtionsaufrufs wird (rekursiv) eine Menge sog. associated namespaces/classes ermittelt, in denen dann die gesuchte Funktion eindeutig gefunden werden muss

```
int main() {
        N::T t;
        t.foo();        // OK: scope durch t festgelegt!
        foo(t);         // wäre ohne ADL fehlerhaft !
                        // dank ADL ok:
        N::foo(t);      // wäre ohne ADL noch akzeptabel
        // nicht aber:
        N::operator<<(std::cout, t); // anstelle von:
        std::cout << t;  // nur mit ADL korrekt
}
```

5x T-Object