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)

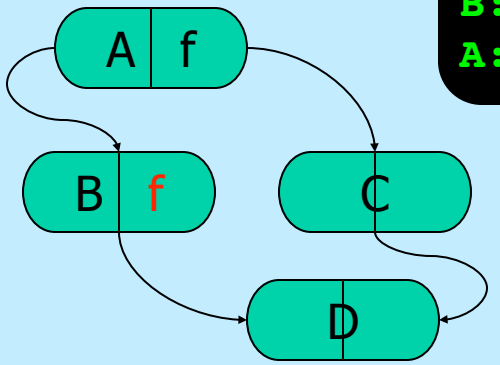

```
class A { public: void f(){cout<<"A::f()\n";}   };
class B: public virtual A {
           public: void f(){cout<<"B::f()\n";}
};
class C: public virtual A {
           public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {};
int main() {   D d;
               // d.f(); ERROR: ambiguous 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()
```

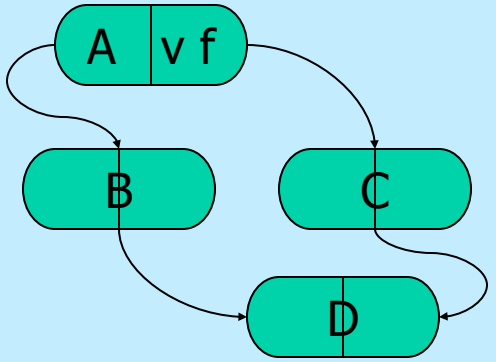


Mehrfachvererbung (multiple inheritance)

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

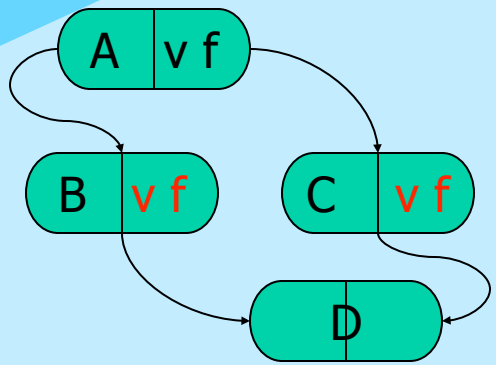
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A { };
class C: public virtual A { };
class D: public B, public C { };
main(){         D d;
                C *pc = &d;
                pc->f();
}
```

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



Mehrfachvererbung (multiple inheritance)

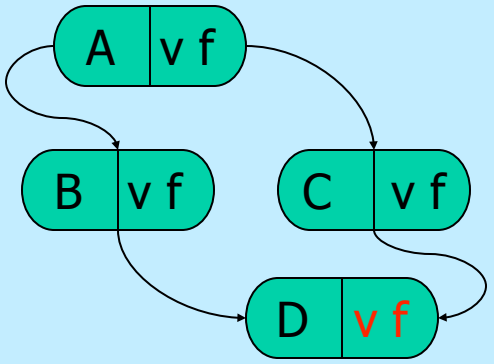
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C { };
// ERROR: no unique final overrider for f() in D
```



Mehrfachvererbung (multiple inheritance)

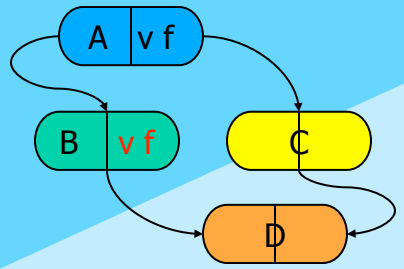
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {
  public: void f(){cout<<"D::f()\n";}
};
... D d; C *pc = &d; pc->f();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

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

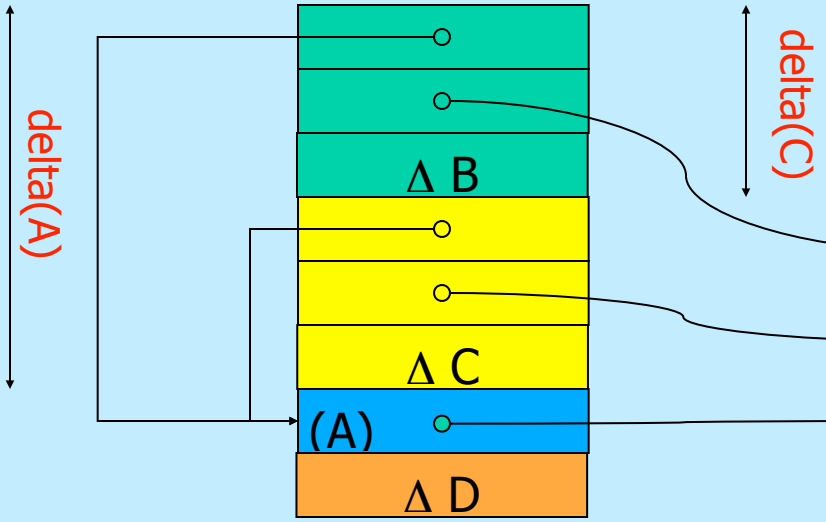


Mehrfachvererbung (multiple inheritance)

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;
```