

## 2. Klassen in C++

### Mehrfachvererbung (*multiple inheritance*)

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

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

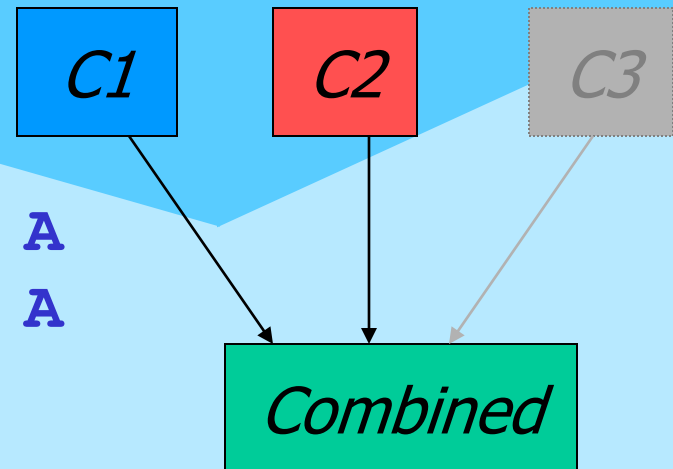
jedes **Combined**-Objekt IST EIN **Concept1** und  
IST EIN **Concept2** (**Concept3**-Abstammung ist ein  
Implementationsdetail)

## 2. Klassen in C++

## 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
```

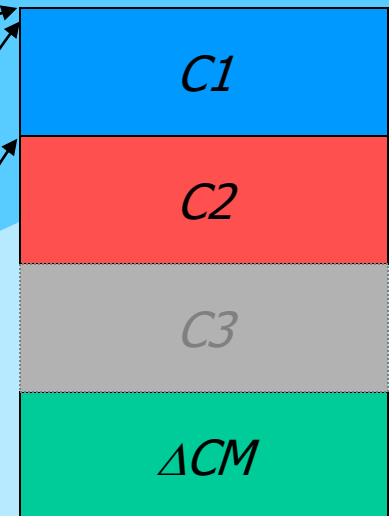


## 2. Klassen in C++

## 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) addresses 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!
```

## 2. Klassen in C++

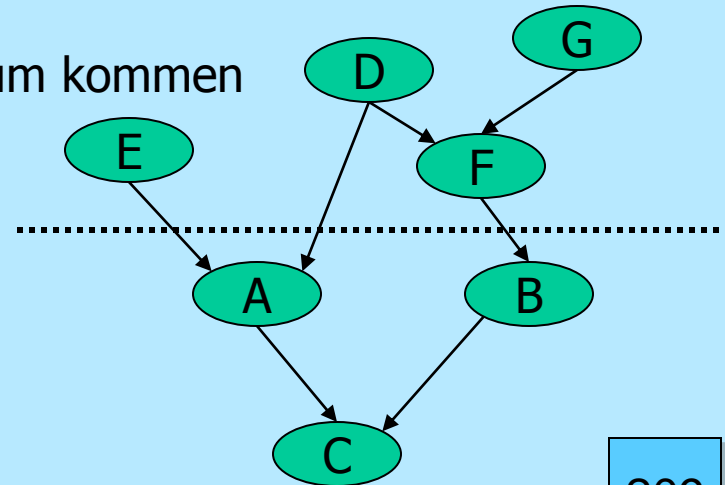
## Mehrfachvererbung (*multiple inheritance*)

eine Klasse kann eine andere nicht direkt mehrfach erben

```
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 {...};
```

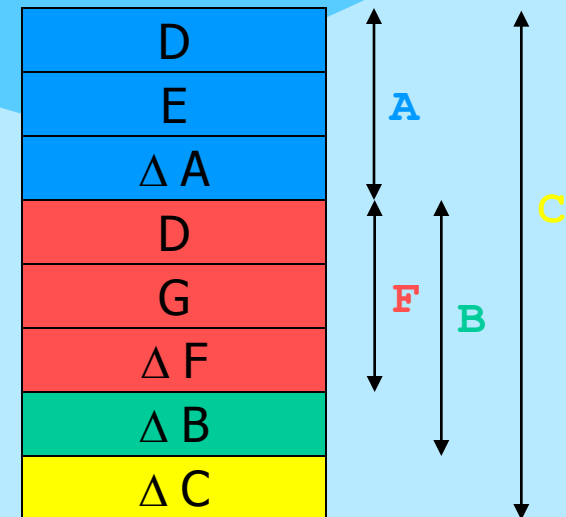


## 2. Klassen in C++

# Mehrfachvererbung (*multiple inheritance*)

ob dabei der mehrfach eingerbte Basiklassenanteil 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 {...};
```

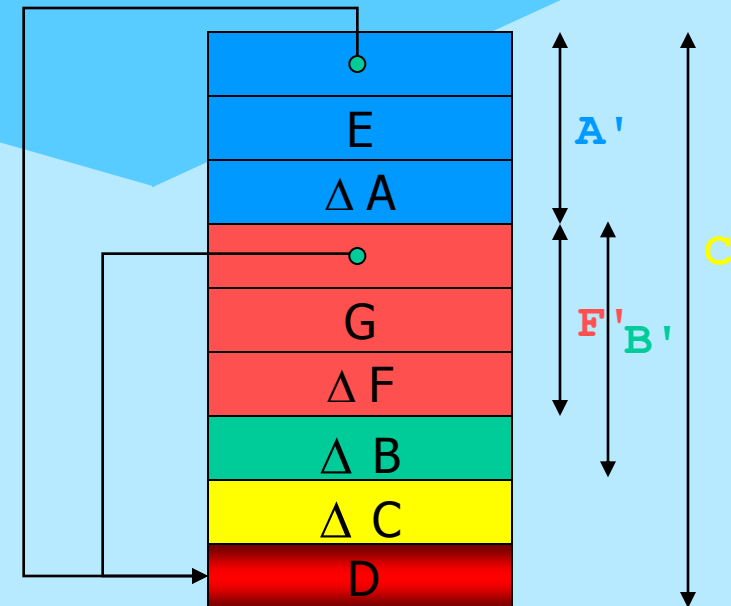
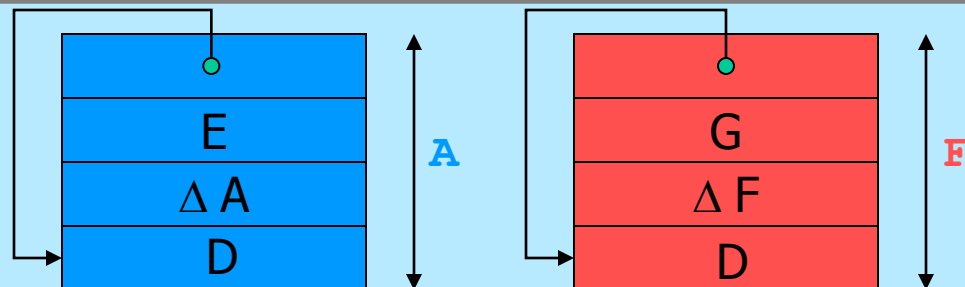


## 2. Klassen in C++

## Mehrfachvererbung (*multiple inheritance*)

ob dabei der mehrfach eingerbte 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 {...};
```



## 2. Klassen in C++

# 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
```

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

*non virtual*

*virtual*

## 2. Klassen in C++

# 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  
};
```



## 2. Klassen in C++

### 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 Members 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
            }
```

## 2. Klassen in C++

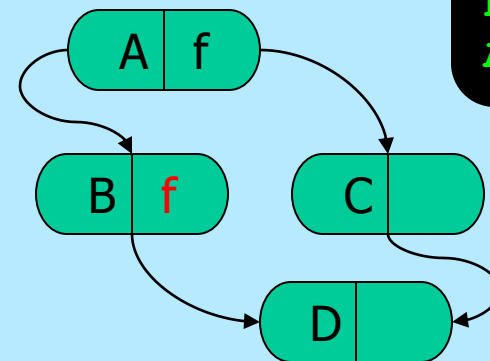
## Mehrfachvererbung (*multiple inheritance*)

Selbst bei virtuellen Basisklassen kann es auf Grund der Maschenbildung sein, dass ein Name eines Members auf mehreren "Wegen" auflösbar ist und zu verschiedenen Members 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()
  
```

## 2. Klassen in C++

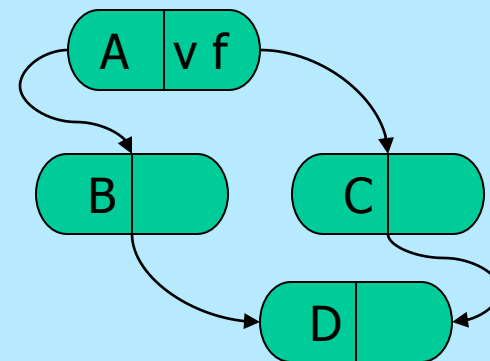
### Mehrfachvererbung (*multiple inheritance*)

Mehrfachvererbung und virtuelle Funktionen sind miteinander kombinierbar, im Falle von virtuellen Basisklassen stehen u.U. ebenfalls mehrere Wege der Auflösung zur Verfügung: fall keine dominante 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()

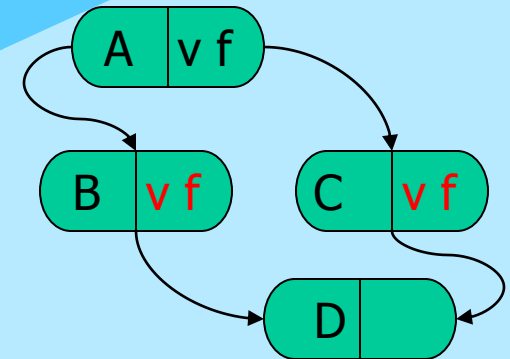


## 2. Klassen in C++

## 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
```

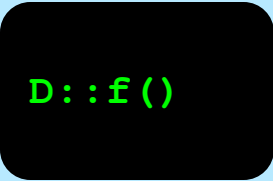


## 2. Klassen in C++

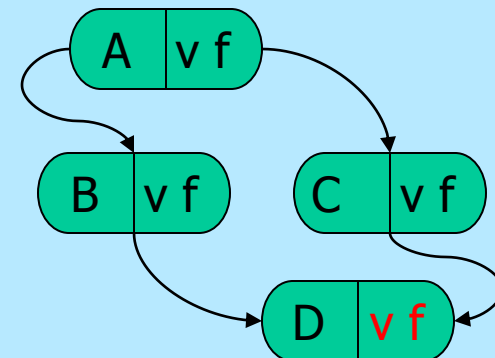
## Mehrfachvererbung (*multiple inheritance*)



```
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(); ...
```



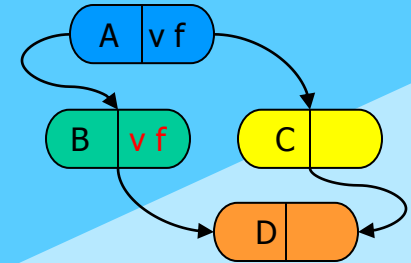
D::f()



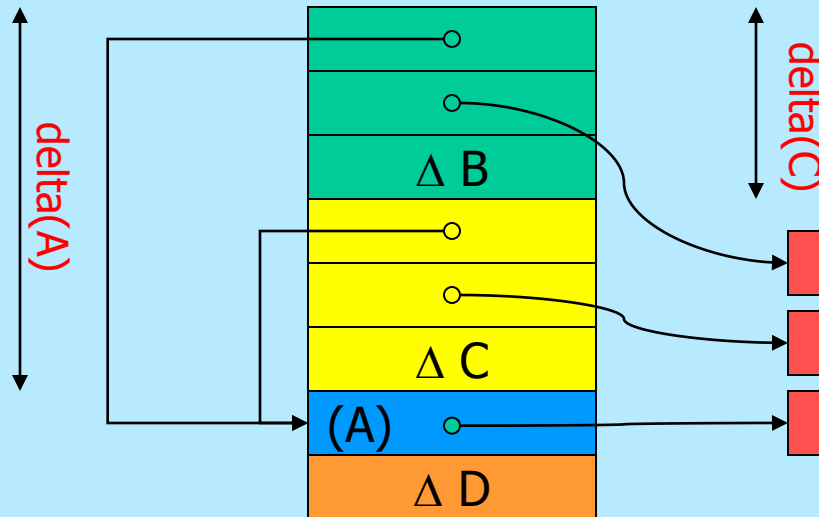
2. Klassen in C++

# 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)

## 2. Klassen in C++

## Mehrfachvererbung (*multiple inheritance*)

Implementation von virtuellen Funktionen wird  
*'slightly more complicated'*

```
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));
```

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

## 2. Klassen in C++

### 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) { .... }
};
```



## 2. Klassen in C++

### 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
    }
};
```

## 2. Klassen in C++

### 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 B: private virtual A {};  
class C: public virtual A {};  
class D: public B, public C {  
    void f() { i++; /* erlaubt, da B: .... public A */ }  
};
```

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

```
-----  
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  
};
```

## 2. Klassen in C++

# 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`, `QString`, `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 !
```

## 2. Klassen in C++

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
```

## 2. Klassen in C++

*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 {
    //...
};
```

## 2. Klassen in C++

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
```

## 2. Klassen in C++

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. Klassen in C++

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;
```



## 2. Klassen in C++

**Achtung**

`using namespace N;` und `using N::name` ( $\forall name \text{ in } N$ )  
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;
}
```

## 2. Klassen in C++

### Achtung

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

`using`- Direktiven sollten nie in unbekanntem 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*) !!

## 2. Klassen in C++

**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

## 2. Klassen in C++

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*/}
```

## 2. Klassen in C++

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



```
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";}  
}
```

*Namensauflösung* erfolgt **immer** in der Reihenfolge

1. lookup
2. overload resolution
3. access check

## 2. Klassen in C++

```
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*)
```

## 2. Klassen in C++

```
// 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
}
```

## 2. Klassen in C++

Lookup **qualifizierter** Namen: im jeweils benannten Scope beginnend rekursiv<sup>(\*)</sup> 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!*



## 2. Klassen in C++

// wie zuvor

```
int main()
```

```
{
```

```
  //  
  //  
  using namespace B;  
  using namespace C;
```

ändert gar nichts!

```
  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");
```

```
}
```

## 2. Klassen in C++

# 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; }
```

## 2. Klassen in C++

### Koenig-Lookup *alias* ADL (argument dependent lookup)

aus allen Parametertypen eines Funktionsaufrufs 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