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
}
```

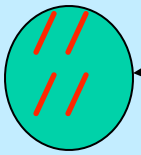
Lookup qualifizierter Namen: im jeweils benannten Scope beginnend rekursiv(* in weiteren bis der Name gefunden wird

(* **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
```

Problem:

Vererbung erlaubt die Wiederverwendung von Programmcode, virtuelle Funktionen ermöglichen dabei den Austausch gewisser Funktionalität

Vererbung erlaubt NICHT die Wiederverwendung durch Parametrisierung von Klassen, z.B.

```
class Stack_of_int{...};
```
und
```
class Stack_of_double{...};
```

NICHT (sinnvoll) realisierbar als:

```
class Stack { ... }; // abstract ??
class Stack_of_int: public Stack { ... };
class Stack_of_double: public Stack { ... };
```

Lösung:

C++ erlaubt die Definition von generischen Klassen:
solche, die von (noch nicht spezifizierten) Typen oder Werten abhängen !

```
// GenericStack.h :
template <class T, int D> // variabel in T und D
class Stack {
protected:
   T *data;
   int top, max;
public:
   Stack(int dim = D): data(new T[dim]), top(0), max(dim) {}
   ~Stack() { delete [] data; } // beide hier inline
   void push (T i); // outline
   T pop();         // outline
};
```

Die Definition von Memberfunktionen außerhalb von generischen Klassen hängt damit selbst von diesen Typen oder Werten ab !

```
// GenericStack.cpp ???
template <class T, int D>
void Stack<T,D>::push (T i) {
      data[top++]=i;
}

template <class T, int D>
T Stack<T,D>::pop () {
      return data[--top];
}
```

eine Template-Klasse definiert ein allgemeines Muster für beliebig viele konkrete Klassen,

aus dem Template selbst lässt sich jedoch (i. allg.) kein Code generieren,

dies geschieht erst bei der sog. Instantiierung der Template-Klasse

```
#include "GenericStack.h"
Stack<int, 10> s1, s2;
Stack<double, 20> s3;
Stack<Stack<int, 10>, 10> ss;
void foo() {
  s1.push(1);  s2.push(2);
  ss.push(s1); ss.push(s2);
  s3.push(3.7);
}
```

Daher muss bei der Instantiierung einer Template-Klasse der Quelltext aller (benutzten) (auch der nicht-inline) Funktionen zur Verfügung stehen:

übliche Verfahrensweise:

```
// GenericStack.h :
template <class T, int D>
class Stack { /* wie oben */ };
#include "GenericStack.cpp"
```

die Instantiierung einer konkreten Klasse geschieht auf explizite Anforderung (erste Verwendung), eine solche kann jedoch weitere Instantiierungen nach sich ziehen

Instantiierung 'auf Vorrat' (ohne sofortige Verwendung) ist möglich:

```
template class Stack<int, 10>;
template class Stack<double, 20>;
template class Stack<Stack<int, 10>, 10>;
```

`<class XYZ>` bedeutet nicht, dass nur mit Klassentypen instantiiert werden darf, vielmehr kündigt `class` einen 'Meta-(Typ-)Parameter' an, alternativ kann hier auch das neue Schlüsselwort `typename` verwendet werden