Weitere (semantische) Besonderheiten bei Templates:

two-phase lookup:

1. **Die Phase in der die Schablone vom Compiler erstmalig inspiziert wird (Template-Parameter sind noch unbekannt): lookup von *non-dependent names* (und zwar nur hier), Basisklassen werden dabei nicht betrachtet – unqualifizierte Namen sind (zumeist) *dependent*!**

2. **Die erneute Inspektion am *point of instantiation* (POI) unter Kenntnis der Template-Parameter. lookup von *dependent names* (per :: nur hier), sonstige dependence in beiden Phasen, ADL wird nur in Phase 2 realisiert.**

Weitere (semantische) Besonderheiten bei Templates:

- 2-phase lookup bringt u.U. unerwartete Effekte

```
template <typename T>
class Base {
  public:
    void exit(int);
};

template <typename T>
class Derived : Base<T> {
  public:
    void foo() {  exit(0); // Base<T>::exit is NOT considered
                  Base<T>::exit(1); // OK, but static bound
                  this->exit(2); // allows for late binding
    }
};
```

leider falsch implementiert in Visual C++ 2008

Weitere (semantische) Besonderheiten bei Templates: **SFINAE**

```
struct Test {
    typedef int Type;
};

template < typename T >
void f(typename T::Type) {} // definition #1

template < typename T >
void f(T) {}                // definition #2

void foo() {
    f<Test>(10); // call #1
    f<int>(10);  // call #2 without error thanks to SFINAE
}
```

Achtung: bei Überladung von Template-Funktionen trifft die explizite Instantiierung keine Auswahl, sondern es werden alle Funktionen 'zugleich' instantiiert von denen die vorliegende Parametersituation eine Funktion eindeutig auswählen muss!

```
template <typename T> int f(T)          { return 1; }
template <typename T> int f(T*)         { return 2; } // kein Spezialisierung !

template <typename T> struct C {
          static int f()                { return 1; }
};
template<typename T> struct C<T*> {
          static int f()                { return 2; }
};

int main() {
          std::cout << f ((int*)0) << std::endl;
          std::cout << f<int> ((int*)0) << std::endl;
          std::cout << f<int*>((int*)0) << std::endl;
          std::cout << C<int>::f() << std::endl;
          std::cout << C<int*>::f() << std::endl;
}
```

```
2
2
1
1
2
```

Der Prozess der Instanziierung ist 'Turing-mächtig'!

Beispiel: Fibonacci-Zahlen berechnen während der Übersetzung

```
#include <iostream>
template <int N>
class Fib {   long val;
public:       Fib():val( Fib<N-1>() + Fib<N-2>() )
{}                          operator long () {return val;}
};
template<>
class Fib<0> {
public:       operator long () {return 1;}
};
```

```
template<>
class Fib<1> {
public:       operator long () {return 1;}
};
int main(){
       std::cout << Fib<42>() << std::endl;
}

// alternativ auch
#include <iostream>
template <int x> int f() { return f<x-2>()+f<x-1>(); }
template <>int f<1>()     { return 1;}
template <>int f<0>()     { return 1;}
int main(){   std::cout<<f<12>()<<std::endl;}
```

Template Klassen Instantiierungen können auch durch Template Funktionen 'provoziert' werden

```
template <class T1, class T2>
struct pair { // so in std !
       T1 first;
       T2 second;
       pair() {}
       pair(const T1& a, const T2& b) : first(a), second(b)
       {}
};
```

```
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;
}

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

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

Type Traits

Generischer Code kennt Typen nicht vorab, möchte u.U. dennoch generalisiert auf Eigenschaften von Typen zugreifen:

```
…
class T1 { // dito T2, T3: classes i have control of
public:
  static const char* someProperty;
};

const char* T1::someProperty = "class T1"; // dito T2, T3

template <class T>
useProperty(T t) { cout << t.someProperty <<endl; }

int main() {
  useProperty(T1());
  useProperty(T2());
  useProperty(T3());
}
```

Type Traits

„Wenn der Typ aber nun ein **int** ist, lieber Heinrich ….“ oder
„Wenn der Typ aber nun nicht meiner ist, lieber Heinrich ….“

Idee: Man stellt jedem Typ einen sog. Traits-Typ (trait == Merkmal, Wesenszug) zur Seite, der diese Eigenschaften bereithält:

```
// voilà:
template <class Any>
class Traits {
public:
      static const char* someProperty;
};
```

Type Traits

... und spezialisiert diesen:

```
template <> class Traits<int> {
public:
        static const char* someProperty;
};

// ACHTUNG: NICHT template<> !
const char* Traits<int>::someProperty = "int";

class T4 { /* without someProperty */ }; // not under my control

template <> class Traits<T4> {
public:
        static const char* someProperty;
};

// ACHTUNG: NICHT template<> !
const char* Traits<T4>::someProperty = "class T4";
```

Type Traits

```
class T5 { // not under my control
public:  static const char* prop;
};

template <> class Traits<T5> {
public:    static const char* someProperty;
};
const char* Traits<T5>::someProperty = T5::prop; // mapping

template <class T>
void useProperty (T t) { Traits<T> tt; cout << tt.someProperty <<endl; }

const char* T5::prop = "class T5";
typedef int T6;

int main() {
      useProperty(T4());
      useProperty(T5());
      useProperty(T6());
}
```

Type Traits

```
// was ist nun mit T1,2,3 ? usеProperty geht nicht mehr wie auf S. 272
// entweder:
template <> struct Traits<T1> {
    static const char* someProperty;
};
const char* Traits<T1>::someProperty=T1::someProperty;

template <> struct Traits<T2> {
        static const char* someProperty;
};
const char* Traits<T2>::someProperty=T2::someProperty;

template <> struct Traits<T3> {
        static const char* someProperty;
};
const char* Traits<T3>::someProperty=T3::someProperty;

// und useProperty von S. 275
```

Type Traits

```
// was ist nun mit T1,2,3 ? usеProperty geht nicht mehr wie auf S. 272
// oder:

template <>
void usеProperty<T1>(T1 t) { cout << t.someProperty <<endl; }

template <>
void usеProperty<T2>(T2 t) { cout << t.someProperty <<endl; }

template <>
void usеProperty<T3>(T3 t) { cout << t.someProperty <<endl; }
```

Die C++ Standardbibliothek basiert wesentlich auf generischem Code

- Ursprünglich als STL (standard template library) bei HP entworfen, bei SGI weiterentwickelt **(http://www.sgi.com/tech/stl/)**
- eine »unkonventionelle« C++ -Bibliothek, bestehend aus Headerfiles, die massiven Gebrauch von Templates machen
  (also keine fertig übersetzten Bibliotheken) !
- auf den ersten Blick im Widerspruch zu einer der Grundgedanken der objektorientierten Programmierung: Trennung von Datenstrukturen und Algorithmen
- stellt elementare Containerklassen für Listen, Vektoren, Mengen, Multimengen, Stacks, Assoziationen ... unter einem einheitlichen Blickwinkel bereit: über allen Typen sind sog. Iteratoren definiert, die sich verhalten, wie Zeiger in die entsprechenden Strukturen, kanonisch in Fortsetzung von C-Feldern und C-Zeigern)
- Zeiger selbst eignen sich auch als elementare Iteratoren

C++ -Standardbibliothek:

die folgenden 51 Standard C++ headers (einschließlich der 18 zusätzlichen Standard C headers) machen eine sog. hosted implementation of Standard C++ aus:

<algorithm>, <bitset>, <cassert>, <cctype>, <cerrno>, <cfloat>,
<ciso646>, <climits>, <clocale>, <cmath>, <complex>, <csetjmp>,
<csignal>, <cstdarg>, <cstddef>, <cstdio>, <cstdlib>,
<cstring>, <ctime>, <cwchar>, <cwctype>, <deque>, <exception>,
<fstream>, <functional>, <iomanip>, <ios>, <iosfwd>, <iostream>,
<istream>, <iterator>, <limits>, <list>, <locale>,
<map>, <memory>, <new>, <numeric>, <ostream>, <queue>, <set>,
<sstream>, <stack>, <stdexcept>, <streambuf>, <string>, <strstream>,
<typeinfo>, <utility>, <valarray>, <vector>

Grundlegende Beobachtung: Algorithmen sind (oft) abstrakt, in dem Sinne, dass sie unabhängig sind von den elementaren Typen sind, auf denen sie operieren:

```
// Beispiel: Lineares Suchen
int* find (int* first, int* last, int value) {
       while (first != last && *first != value) ++first;
       return first;
}  //                          warum nicht bool find(....) ???
```

das geht genauso mit double*, XYZ*, ... : Abstraktion vom elementaren Typ und Reduzierung auf Konzepte von Anfang, Ende, Vergleich ...

```
template <class InputIterator, class T>
InputIterator find
(InputIterator first, InputIterator last, const T& value) {
       while (first != last && *first != value) ++first;
       return first;
}
```

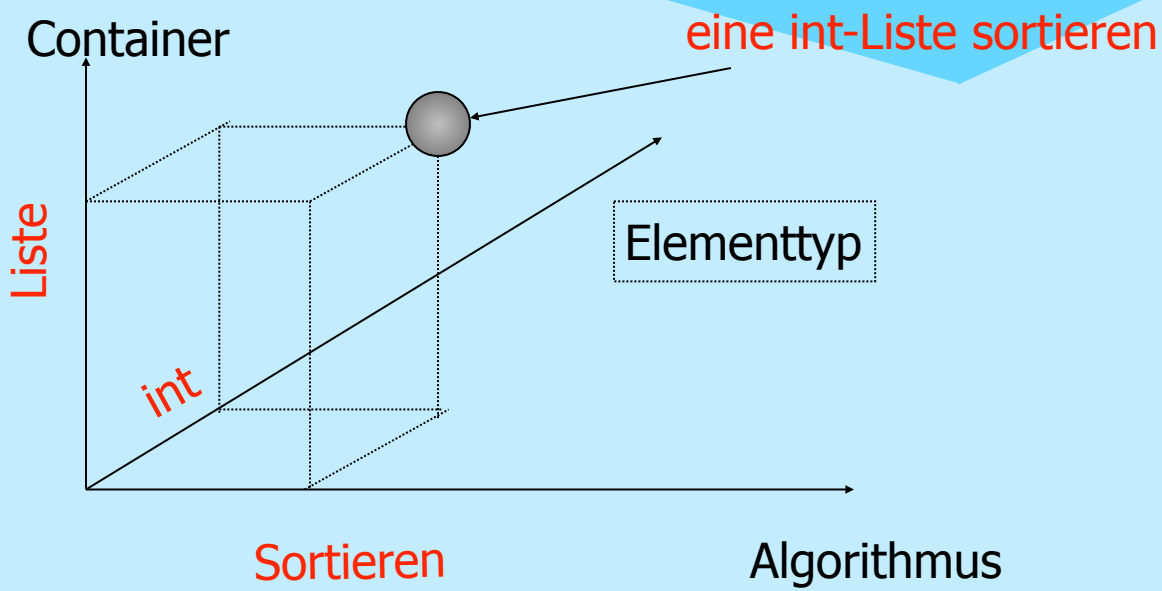
Weitere Verallgemeinerungen sind oft möglich:

```
template <class InputIterator, class Predicate>
InputIterator find_if
(InputIterator first, InputIterator last, Predicate pred) {
      while (first != last && !pred(*first)) ++first;
      return first;
}
// pred ist Verallgemeinerung von ==(. , value)
// pred muss als Funktion aufrufbar sein
```

Ziel: "Algorithmus x angewendet auf Containertyp y des Basistyps z "

Traditioneller Ansatz:

m (Algorithmen) * n (Containertypen) * t (Basistypen)



- Templates in ihrer primären Anwendung:
  m (Algorithmen) * n (template<t> Container)
- Ansatz der STL: die Algorithmen agieren auf austauschbaren (aus Templates instantiierbaren) Zugriffsoperationen
  m (Algorithmen) + n (template<t> Container)
- Iteratoren entkoppeln die Containertypen von den Algorithmen, die auf ihnen operieren
- ein Iteratorwert verkörpert eine Position, kann mittels ++ weitergesetzt werden ACHTUNG: man benutze immer Prefix-In/Dekrement (keine temporäre Variable)
- Iteratoren können auf Un-/Gleichheit geprüft werden
- * liefert das referenzierte Element (wie bei Zeigern)
- (Input-)Iteratorobjekte können kopiert werden !