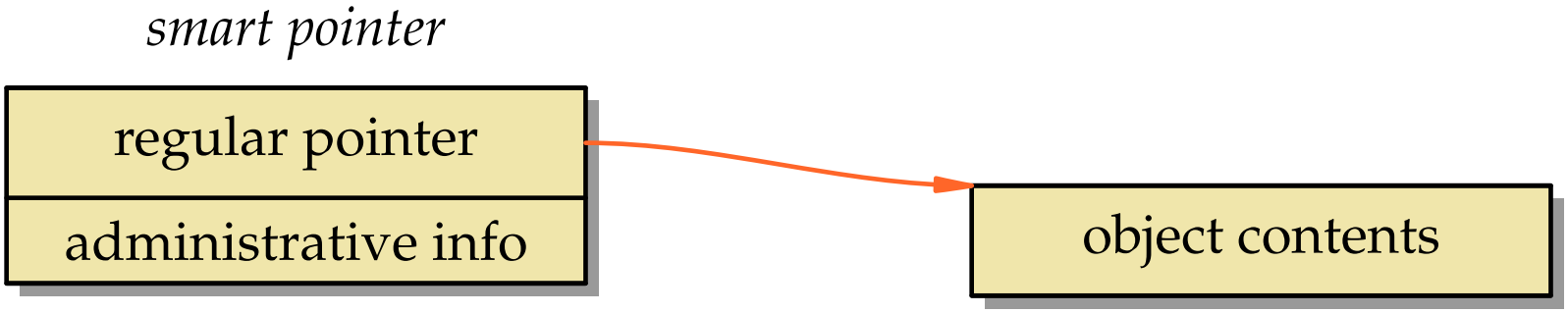


- Intelligente Zeiger (*smart pointers*) entsprechen weitgehend normalen Zeigern, haben aber Sonderfunktionalitäten aufgrund weiterer Verwaltungsinformationen.
- Sie werden insbesondere dort eingesetzt, wo die Sprache selbst keine Infrastruktur für die automatisierte Speicherfreigabe anbietet.
- Sie unterstützen das RAII-Prinzip für Zeiger.

- Seit dem C++11-Standard sind intelligente Zeiger Bestandteil der C++-Bibliothek. Zuvor gab es nur den inzwischen abgelösten *auto_ptr* und die Erweiterungen der Boost-Library, die jetzt praktisch übernommen worden sind.
- C++11 bietet folgende Varianten an:

unique_ptr	nur ein Zeiger auf ein Objekt
shared_ptr	mehrere Zeiger auf ein gemeinsames Objekt mit externem Referenzzähler
weak_ptr	nicht das Überleben sichernder „schwacher“ Zeiger auf ein Objekt mit externem Referenzzähler

- Grundsätzlich sollte ein mit **new** erzeugtes Objekt mit **delete** wieder freigegeben werden, sobald der letzte Verweis entfernt wird.
- Unterbleibt dies, haben wir ein Speicherleck.
- Wichtig ist aber auch, dass kein Objekt mehrfach freigegeben wird. Dies kann bei manueller Freigabe leicht geschehen, wenn es mehrere Zeiger auf ein Objekt gibt.
- Intelligente Zeiger können sich auch dann um eine korrekte Freigabe kümmern, wenn eine Ausnahmenbehandlung ausgelöst wird.
- Jedoch können zyklische Datenstrukturen mit der Verwendung von Referenzzählern alleine nicht korrekt aufgelöst werden. Hier sind ggf. Ansätze mit sogenannten „schwachen“ Zeigern denkbar.

- Auf ein Objekt sollten nur Zeiger eines Typs verwendet werden.
- Die einzige Ausnahme davon ist die Mischung von *shared_ptr* und *weak_ptr*.
- Im Normalfall bedeutet dies, dass die entsprechenden Klassen angepasst werden müssen, da es dann nicht mehr zulässig ist, **this** zurückzugeben.
- Üblicherweise sollte sogleich bei dem Entwurf einer Klasse geplant werden, welche Art von Zeigern zum Einsatz kommt.
- Normalerweise sollten entsprechend des RAII-Prinzips „nackte“ Zeiger außerhalb isolierter Fälle konsequent vermieden werden.

- Wenn es nur einen einzigen Zeiger auf ein Objekt geben soll, dann empfiehlt sich die Verwendung von *unique_ptr*.
- Das ist besonders geeignet für lokale Zeigervariablen oder Zeiger innerhalb einer Klasse.
- Die Freigabe erfolgt dann vollautomatisch, sobald der zugehörige Block bzw. das umgebende Objekt freigegeben werden.
- Bei einer Zuweisung wird der Besitz des Zeigers übertragen. Das funktioniert nur entsprechend mit einem sogenannten *move assignment*, d.h. der Zeigerwert wird von einem anderen *unique_ptr*-Objekt gerettet, der im nächsten Moment ohnehin dekonstruiert wird.
- Andere Zuweisungen dieser Zeiger sind nicht möglich, da dies die Restriktion des exklusiven Zugangs verletzen würde.

ptrex.cpp

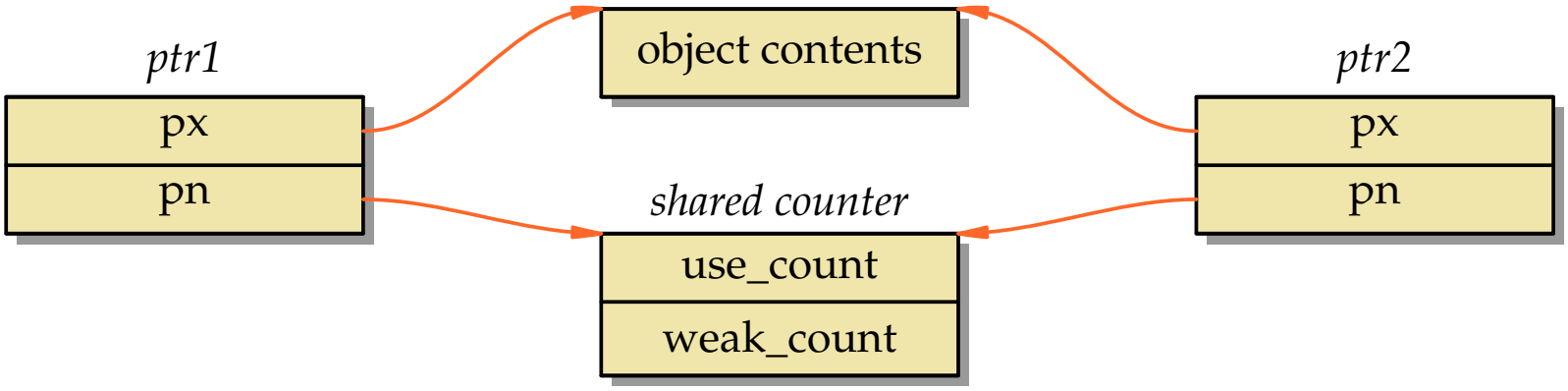
```
void f(int i) {
   Object* ptr = new Object(i);
   if (i == 2) {
      throw something();
   }
   delete ptr;
}
```

- Wenn Objekte in einer Funktion nur lokal erzeugt und verwendet werden, ist darauf zu achten, dass die Freigabe nicht vergessen wird.
- Dies passiert jedoch leicht bei Ausnahmenbehandlungen (möglicherweise durch eine aufgerufene Funktion) oder bei frühzeitigen **return**-Anweisungen.

```
void f(int i) {
   std::unique_ptr<Object> ptr(new Object(i));
   if (i == 2) {
      throw something();
   }
}
```

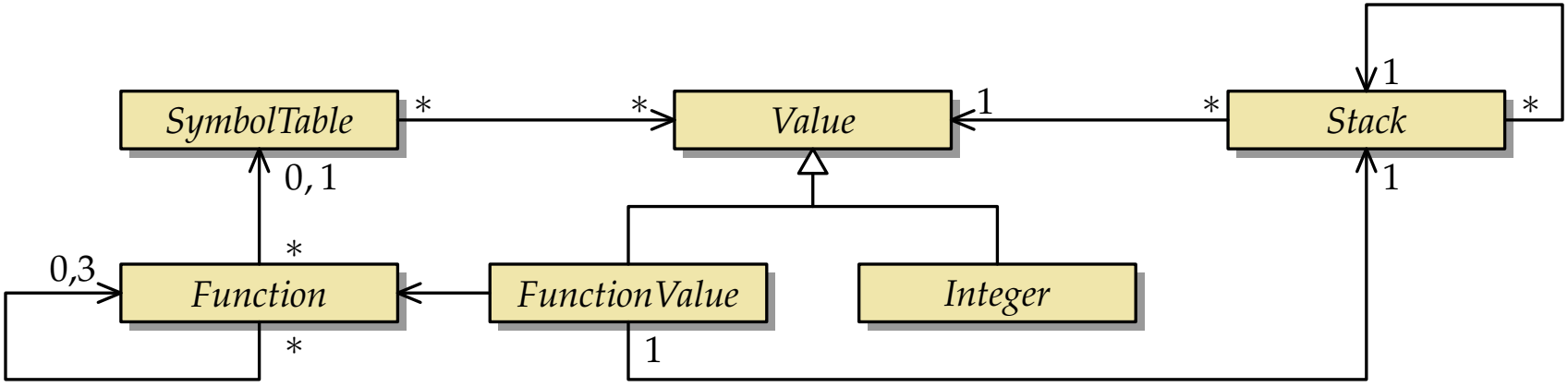
- *ptr* kann hier wie ein normaler Zeiger verwendet werden, abgesehen davon, dass eine Zuweisung an einen anderen Zeiger nicht zulässig ist.
- Dann erfolgt die Freigabe des Objekts vollautomatisch über den Dekonstruktor.
- Beginnend ab C++14 gibt es die Funktion *std::make_unique*, die im Vergleich zur früheren Variante *exception safeness* anbietet:

```
void f(int i) {
   auto ptr = std::make_unique<Object>(i);
   if (i == 2) {
      throw something();
   }
}
```



- Für den allgemeinen Einsatz empfiehlt sich die Verwendung von *shared_ptr*, das mit Referenzzählern arbeitet.
- Zu jedem referenzierten Objekt gehört ein intern verwaltetes Zählerobjekt, das die Zahl der Verweise zählt. Sobald *use_count* auf 0 sinkt, erfolgt die Freigabe des Objekts.
- Das Zählerobjekt wird erst freigegeben, wenn neben *use_count* auch *weak_count* auf 0 sinkt.
- Es ist darauf zu achten, dass für jedes Objekt nur ein gemeinsames Zählerobjekt existiert.

Beispiel: Kleine Sprache mit Lambda-Ausdrücken 367



- Im 7. Übungsblatt hatten wir eine kleine Sprache mit λ-Ausdrücken.
- Die zugehörige Datenstruktur hat den varianten Datentyp *Value*. Funktionswerte (*FunctionValue*) bestehen aus einem Zeiger auf den Kaktusstack (*closure*) und einen Zeiger auf das konstruierte Funktionsobjekt, das (im Falle einer *if*-Anweisung) auf bis zu drei weitere Funktionsobjekte verweisen kann.
- Die Datenstruktur ist komplex und es lässt sich nicht trivial feststellen, wann der letzte Verweis auf ein Objekt wegfällt.

types.hpp

```
class Value;
using ValuePtr = std::shared_ptr<Value>;

class Stack;
using StackPtr = std::shared_ptr<Stack>;

using Function = std::function<ValuePtr(StackPtr)>;
using FunctionPtr = std::shared_ptr<Function>;
```

- Bei zyklischen Typreferenzen ist es sinnvoll, alle Klassen zuerst zu deklarieren.
- Dann können auch sofort die entsprechenden intelligenten Zeigertypen definiert werden.

stack.hpp

```
class Stack {
   public:
      Stack(StackPtr next, ValuePtr value) :
         next(next), value(value), len(next? next->len+1: 1) {
      }
      ValuePtr operator[](unsigned int index) const {
         assert(index < len);
         if (index == 0) return value;
         return (*next)[index-1];
      }
   private:
      ValuePtr value;
      StackPtr next;
      unsigned int len;
};
```

- Statt *Value*∗ oder *Stack*∗ wird hier konsequent *ValuePtr* bzw. *StackPtr* eingesetzt.

value.hpp

```
class Value {
   public: virtual ~Value() {};
};

class Integer: public Value { /* ... */ };
using IntegerPtr = std::shared_ptr<Integer>;

class FunctionValue : public Value { /* ... */ };
using FunctionValuePtr = std::shared_ptr<FunctionValue>;
```

- Statt der varianten Klasse könnte die Implementierung auch eine Typenhierarchie vorsehen.
- Die Kompatibilität innerhalb der *Value*-Hierarchie überträgt sich auch auf die zugehörigen intelligenten Zeiger. Zwar bilden die intelligenten Zeigertypen keine formale Hierarchie, aber sie bieten Zuweisungs-Operatoren auch für fremde Datentypen an, die nur dann funktionieren, wenn die Kompatibilität für die entsprechenden einfachen Zeigertypen existiert.

parser.cpp

```
FunctionPtr Parser::parseExpression() {
   // ...
   // expr --> integer
   if (getToken().symbol == Token::INTEGER) {
      int integer = getToken().integer;
      nextToken();
      return std::make_shared<Function>([=] (StackPtr sp) {
         return std::make_shared<Integer>(integer);
      });
   }
   // ...
}
```

- *make_shared* erzeugt ein Objekt des angegebenen Typs mit **new** und liefert den passenden intelligenten Zeigertyp zurück.
- Das ist in diesem Beispiel *shared_ptr<Integer>*, das entsprechend der Klassenhierarchie an den allgemeinen Zeigertyp *FunctionPtr* zugewiesen werden kann.

lambda.cpp

```
FunctionPtr f;
while (f = parser.getFunction()) {
   ValuePtr value = (*f)(nullptr);
   auto intval = std::dynamic_pointer_cast<Integer>(value);
   if (intval) {
      std::cout << intval->get_integer() << std::endl;
   }
}
```

- Statt **dynamic_cast** ist bei intelligenten Zeigern *dynamic_pointer_cast* zu verwenden, um eine ungewollte Neu-Erzeugung eines Zählerobjekts zu vermeiden.
- Genauso wie bei **dynamic_cast** wird ein Nullzeiger geliefert, falls der angegebene Zeiger nicht den passenden Typ hat.

```
theon$ cd lambda
theon$ time lambda <primes.lambda
541

real    0m0.330s
user    0m0.322s
sys     0m0.004s
theon$ cd ../lambda-hier
theon$ time lambda <primes.lambda
541

real    0m0.371s
user    0m0.359s
sys     0m0.006s
theon$
```

- Genauso wie **dynamic_cast** ist auch *std*::*dynamic_pointer_cast* nicht ohne Kosten.
- Variante Klassen können daher von Vorteil sein, wenn klar ist, dass eine Erweiterung der Vielfalt nicht vorgesehen ist.
- Variante Objekte benötigen aber mehr Speicher, da immer das Maximum zum Zuge kommt. Im Beispiel: 122 MB vs. 110 MB.

- Bei Referenzzyklen bleiben die Referenzzähler positiv, selbst wenn der Zyklus insgesamt nicht mehr von außen erreichbar ist.
- Eine automatisierte Speicherfreigabe (*garbage collection*) würde den Zyklus freigeben, aber mit Zeigern auf Basis von *shared_ptr* gelingt dies nicht.
- Eine Lösung für dieses Problem sind sogenannte schwache Zeiger (*weak pointers*), die bei der Referenzzählung nicht berücksichtigt werden.

Beispiel: Doppelt verkettete Liste

list.hpp

```
template <typename T>
class List {
   private:
      struct Element;
      using Link = std::shared_ptr<Element>;
      using WeakLink = std::weak_ptr<Element>;
      struct Element {
         Element(const T& elem) : elem(elem) { }
         T elem;
         Link next;
         WeakLink prev;
      };
      Link head;
      Link tail;
   public:
      class Iterator {
         // ...
      };
      // ...
};
```

Beispiel: Doppelt verkettete Liste

list.hpp

```
using Link = std::shared_ptr<Element>;
using WeakLink = std::weak_ptr<Element>;
struct Element {
   Element(const T& elem) : elem(elem) { }
   T elem;
   Link next;
   WeakLink prev;
};
```

- Die einzelnen Glieder einer doppelt verketteten Liste verweisen jeweils auf den Nachfolger und den Vorgänger.
- Wenn mindestens zwei Glieder in einer Liste enthalten ist, ergibt dies eine zyklische Datenstruktur.
- Das kann dadurch gelöst werden, dass für die Rückverweise schwache Zeiger verwendet werden.

list.hpp

```
void push_back(const T& object) {
   Link ptr = std::make_shared<Element>(object);
   ptr->prev = tail;
   if (head) {
      tail->next = ptr;
   } else {
      head = ptr;
   }
   tail = ptr;
}
```

- Eine Zuweisung eines *shared_ptr* an den korrespondierenden *weak_ptr* ist problemlos möglich wie hier bei: $ptr->prev = tail$

list.hpp

```
class Iterator {
   public:
      class Exception: public std::exception {
         // ...
      };
      bool valid() const { /* ... */ }
      T& operator*() { /* ... */ }
      Iterator& operator++() { /* ... */ }
      Iterator operator++(int) { /* ... */ }
      Iterator& operator--() { /* ... */ }
      Iterator operator--(int) { /* ... */ }
      bool operator==(const Iterator& other) { /* ... */ }
      bool operator!=(const Iterator& other) { /* ... */ }
   private:
      friend class List;
      Iterator() {}
      Iterator(WeakLink ptr) : ptr(ptr) {}
      WeakLink ptr;
};
```

list.hpp

```
T& operator*() {
   Link p = ptr.lock();
   if (p) {
      return p->elem;
   } else {
      throw Exception("iterator is expired");
   }
}
```

- Ein schwacher Zeiger kann mit Hilfe der *lock*-Methode in einen regulären Zeiger verwandelt werden.
- Wenn das referenzierte Objekt mittlerweile freigegeben wurde, erhalten wir das Äquivalent eines **nullptr**.

Vergleich von schwachen Zeigern

list.hpp

```
bool operator==(const Iterator& other) {
   Link p1 = ptr.lock();
   Link p2 = other.ptr.lock();
   return p1 == p2;
}
bool operator!=(const Iterator& other) {
   return !(*this == other);
}
```

- Schwache Zeiger können erst dann miteinander verglichen werden, wenn sie zuvor in reguläre Zeiger konvertiert werden.
- Nullzeiger werden hier als äquivalent angesehen.

```
#include <memory>

class Object;
typedef std::shared_ptr<Object> ObjectPtr;
class Object: public std::enable_shared_from_this<Object> {
   public:
      ObjectPtr me() {
         return shared_from_this();
      }
};
```

- Die Grundregel, dass auf ein Objekt nur Zeiger eines Typs verwendet werden sollten, stößt auf ein Problem, wenn statt **this** ein passender intelligenter Zeiger zurückzugeben ist.
- Eine Lösung besteht darin, die Klasse von *std*::*enable_shared_from_this* abzuleiten. Dann steht die Methode *shared_from_this* zur Verfügung. Dies wird implementiert, indem im Objekt zusätzlich ein schwacher Zeiger auf das eigene Objekt verwaltet wird.

```
#include <memory>

class Object;
using ObjectPtr = std::shared_ptr<Object>;
class Object {
   public:
      class Key {
         friend class Object;
         Key() {}
      };
      static ObjectPtr create() {
         return std::make_shared<Object>(Key());
      }
      Object(Key&& key) {}
};
```

- Um die Grundregel durchzusetzen, erscheint es gelegentlich sinnvoll, die regulären Konstruktoren zu verbergen.
- **private** dürfen Sie jedoch nicht sein, da *std*::*make_shared* einen passenden öffentlichen Konstruktor benötigt.
- Eine Lösung bietet der *pass key*-Ansatz. Der Konstruktor ist zwar öffentlich, aber ohne privaten Schlüssel nicht benutzbar.

```
std::shared_ptr<int[]> p(new int[10]);
p[7] = 42;
```

- *std*::*shared_array* kann nicht ohne weiteres mit Arrays verwendet werden, da sich die Operatoren **new** und **delete** jeweils davon abhängen, ob es sich um Arrays handelt oder nicht.
- Erst ab C++17 wurde *std*::*shared_array* dahingehend erweitert, dass auch Arrays unterstützt werden. Hinzugekommen ist der Index-Operator.
- Das dazu passende *std*::*make_shared* ist für C++20 geplant.

Die Standard-Template-Library (STL) bietet eine Reihe von Template-Klassen für Container, eine allgemeine Schnittstelle für Iteratoren und eine Sammlung von Algorithmen an:

- Iteratoren sind eine Verallgemeinerung von Zeigern und dienen als universelle Schnittstelle für den Zugriff auf eine Sequenz von Objekten.
- Zu den Algorithmen gehören Abfragen auf Sequenzen, die die Objekte nicht verändern (diverse Suchen, Vergleiche), solche die sie verändern (Kopieren, Verschieben, Transformieren) und sonstige Operationen (Sortieren, binäre Suche, Mengen-Operationen auf sortieren Sequenzen). Die Algorithmen arbeiten allesamt mit Iteratoren und sichern wohldefinierte Komplexitäten zu unabhängig von den verwendeten Datenstrukturen.
- Container bieten eine breite Vielfalt an Datenstrukturen, um Objekte zu beherbigen. Dazu gehören u.a. Arrays, lineare Listen, sortierte balancierte binäre Bäume und Hash-Verfahren.

Implementierungstechnik	Name der Template-Klasse	
Lineare Listen	*list*	*forward_list*
Dynamische Arrays	*vector*	*string*
	deque	
Adapter	*stack*	*queue*
Balancierte binäre	*set*	*multiset*
sortierte Bäume	*map*	*multimap*
Hash-Verfahren	*unordered_set*	*unordered_multiset*
	unordered_map	*unordered_multimap*

- All die genannten Container-Klassen mit Ausnahme der *unordered*-Varianten besitzen eine Ordnung. Eine weitere Ausnahme sind hier noch die Template-Klassen *multiset* und *multimap*, die keine definierte Ordnung für mehrfach vorkommende Schlüssel haben.
- Die Unterstützung von Hash-Tabellen (*unordered_map* etc.) ist erst mit C++11 gekommen. Zuvor sah die Standard-Bibliothek keine Hash-Tabellen vor.
- Gelegentlich gab es früher den Standard ergänzende Bibliotheken, die dann andere Namen wie etwa *hash_set*, *hash_map* usw. hatten.

Eine gute Container-Klassenbibliothek strebt nach einer übergreifenden Einheitlichkeit, was sich auch auf die Methodennamen bezieht:

Methode	Beschreibung
begin()	liefert einen Iterator, der auf das erste Element verweist
end()	liefert einen Iterator, der hinter das letzte Element zeigt
rbegin()	liefert einen rückwärts laufenden Iterator, der auf das letzte Element verweist
rend()	liefert einen rückwärts laufenden Iterator, der vor das erste Element zeigt
empty()	ist wahr, falls der Container leer ist
size()	liefert die Zahl der Elemente
clear()	leert den Container
erase(*it*)	entfernt das Element aus dem Container, auf das *it* zeigt

Methode	Beschreibung
emplace()	erwartet bei sequentiellen Containern einen Iterator und die Parameter für einen Konstruktor des Elementtyps. Das Objekt wird dann innerhalb des Containers *vor* der Position des Iterators plaziert. Bei assoziativen Containern werden nur die Parameter für den Konstruktor angegeben.
emplace_hint()	erlaubt bei sortierten assoziativen Containern die Angabe eines Iterators, der ggf. die Suche nach der richtigen Stelle vereinfacht.

Methode	Beschreibung
try_emplace()	Analog zu *emplace*() und *emplace_hint*() mit expliziter Angabe des Schlüssels bei assoziativen Containern. Wenn es bereits ein Objekt mit dem Schlüssel gibt, wird das Objekt nicht konstruiert. (Bei *emplace* wird im Konfliktfall das Objekt zuerst konstruiert und dann zerstört.)
extract()	Erlaubt das Umhängen von Objekten ohne Verschieben oder Kopieren von einem Container zu einem anderen.
merge()	Umhängen aller Objekte eines Containers in einen anderen.

Methode	Beschreibung	unterstützt von
front()	liefert das erste Element eines Containers	*vector*, *list*, *deque*
back()	liefert das letzte Element eines Containers	*vector*, *list*, *deque*
push_front()	fügt ein Element zu Beginn ein	*list*, *deque*
push_back()	hängt ein Element an das Ende an	*vector*, *list*, *deque*
pop_front()	entfernt das erste Element	*list*, *deque*
pop_back()	entfernt das letzte Element	*vector*, *list*, *deque*
[*n*]	liefert das *n*-te Element	*vector*, *deque*
at(*n*)	liefert das *n*-te Element mit Index-Überprüfung zur Laufzeit	*vector*, *deque*

Listen gibt es in zwei Varianten: *std*::*list* ist doppelt verkettet und wird überwiegend verwendet. Wenn der Speicherverbrauch minimiert werden soll, kann die einfach verkettete *std*::*forward_list* verwendet werden, die aber nicht mehr alle Vorteile der regulären Liste bietet:

Vorteile:

- Überall konstanter Aufwand beim Einfügen und Löschen. (Dies schließt nicht das Finden eines Elements in der Mitte ein.)
- Unterstützung des Zusammenlegens von Listen, des Aufteilens und des Umdrehens.

Nachteile:

- Kein indizierter Zugriff. Entsprechend ist der Suchaufwand linear.

Vorteile:

- Schneller indizierter Zugriff (theoretisch kann dies gleichziehen mit den eingebauten Arrays).
- Konstanter Aufwand für Einfüge- und Löschoperationen am Ende. (Beim Einfügen kann es aber Ausnahmen geben, siehe unten.)
- Geringerer Speicherverbrauch, weil es keinen Overhead für einzelne Elemente gibt.
- Cache-freundlich, da die Elemente des Vektors zusammenhängend im Speicher liegen.

Nachteile:

- Da der belegte Speicher zusammenhängend ist, kann eine Vergrößerung eines Vektors zu einer Umkopieraktion führen mit linearem Aufwand.
- Weder *push_front* noch *pop_front* werden unterstützt.

Vorteile:

- Erlaubt indizierten Zugriff in konstanter Zeit
- Einfüge- und Lösch-Operationen an den Enden mit konstanten Aufwand.

Nachteile:

- Einfüge- und Lösch-Operationen in der Mitte haben einen linearen Aufwand.
- Kein Aufteilen, kein Zusammenlegen (im Vergleich zu Listen).
- Erheblich erhöhter Speicheraufwand im Vergleich zu einem Vektor, da es sich letztlich um einen Vektor von Vektoren handelt.
- Der indizierte Aufwand ist zwar konstant, hat aber eine Indirektion mehr als beim Vektor.
- Eine Deque ist somit ineffizienter als ein Vektor oder eine Liste auf deren jeweiligen Paradedisziplinen. Sie ist nur sinnvoll, wenn der indizierte Zugriff und beidseitiges Einfügen und Löschen wichtig sind.

std::*queue* und *std*::*stack* basieren auf einem anderen Container-Typ (zweiter Template-Parameter, *std*::*deque* per Voreinstellung) und bieten dann nur die entsprechende Funktionalität an:

Operation	Rückgabe-Typ	Beschreibung
empty()	**bool**	liefert *true*, falls der Container leer ist
size()	*size_type*	liefert die Zahl der enthaltenen Elemente
top()	*value_type*&	liefert das letzte Element; eine **const**-Variante wird ebenfalls unterstützt
push(*element*)	**void**	fügt ein Element hinzu
pop()	**void**	entfernt ein Element

Es gibt vier sortierte assoziative Container-Klassen in der STL:

	Schlüssel/Werte-Paare	Nur Schlüssel
Eindeutige Schlüssel	*map*	*set*
Mehrfache Schlüssel	*multimap*	*multiset*

- Der Aufwand der Suche nach einem Element ist logarithmisch.
- Kandidaten für die Implementierung sind AVL-Bäume oder Red-Black-Trees.
- Voreinstellungsgemäß wird $<$ für Vergleiche verwendet, aber es können auch andere Vergleichs-Operatoren spezifiziert werden. Der $==$-Operator wird nicht verwendet. Stattdessen wird die Äquivalenzrelation von $<$ abgeleitet, d.h. a und b werden dann als äquivalent betrachtet, falls $!(a < b) \&\& !(b < a)$.
- Alle assoziativen Container haben die Eigenschaft gemeinsam, dass vorwärts laufende Iteratoren die Schlüssel in monotoner Reihenfolge entsprechend des Vergleichs-Operators durchlaufen. Im Falle von Container-Klassen, die mehrfach vorkommende Schlüssel unterstützen, ist diese Reihenfolge nicht streng monoton.

- Assoziative Container mit eindeutigen Schlüsseln akzeptieren Einfügungen nur, wenn der Schlüssel bislang noch nicht verwendet wurde.
- Im Falle von *map* und *multimap* ist jeweils ein Paar, bestehend aus einem Schlüssel und einem Wert zu liefern. Diese Paare haben den Typ *std*::*pair*<**const** *Key*, *Value*>, der dem Typ *value_type* der instanziierten Template-Klasse entspricht.
- Der gleiche Datentyp für Paare wird beim Dereferenzieren von Iteratoren bei *map* und *multimap* geliefert.
- Das erste Feld des Paares (also der Schlüssel) wird über den Feldnamen *first* angesprochen; das zweite Feld (also der Wert) ist über den Feldnamen *second* erreichbar.

- Die Template-Klasse *map* unterstützt den []-Operator, der den Datentyp für Paare vermeidet, d.h. Zuweisungen wie etwa *mymap*[*key*] = *value* sind möglich.
- Jedoch ist dabei Vorsicht geboten: Es gibt keine **const**-Variante des []-Operators und ein Zugriff auf *mymap*[*key*] führt zum Aufruf des Default-Konstruktors für das Element, wenn es bislang noch nicht existierte. Entsprechend ist der []-Operator nicht zulässig in **const**-Methoden und stattdessen erfolgt der Zugriff über einen *const_iterator*.

Methode	Beschreibung
insert(*t*)	Einfügen eines Elements: *std*::*pair*<*iterator*,**bool**> wird von *map* und *set* geliefert, wobei der Iterator auf das Element mit dem Schlüssel verweist und der **bool**-Wert angibt, ob die Einfüge-Operation erfolgreich war oder nicht. Bei *multiset* und *multimap* wird nur ein Iterator auf das neu hinzugefügte Element geliefert.
insert(*it*,*t*)	Analog zu *insert*(*t*). Falls das neu einzufügende Element sich direkt hinter *t* einfügen lässt, erfolgt die Operation mit konstantem Aufwand.
erase(*k*)	Entfernt alle Elemente mit dem angegebenen Schlüssel.
erase(*it*)	Entfernt das Element, worauf *it* zeigt.
erase(*it1*, *it2*)	Entfernt alle Elemente aus dem Bereich $[it1, it2)$.

Methode	Beschreibung
find(k)	Liefert einen Iterator, der auf ein Element mit dem gewünschten Schlüssel verweist. Falls es keinen solchen Schlüssel gibt, wird *end()* zurückgeliefert.
count(k)	Liefert die Zahl der Elemente mit einem zu k äquivalenten Schlüssel. Dies ist insbesondere bei *multimap* und *multiset* sinnvoll.
lower_bound(k)	Liefert einen Iterator, der auf das erste Element verweist, dessen Schlüssel nicht kleiner als k ist.
upper_bound(k)	Liefert einen Iterator, der auf das erste Element verweist, dessen Schlüssel größer als k ist.

Es gibt vier unsortierte assoziative Container-Klassen in der STL:

	Schlüssel/Werte-Paare	Nur Schlüssel
Eindeutige Schlüssel	*unordered_map*	*unordered_set*
Mehrfache Schlüssel	*unordered_multimap*	*unordered_multiset*

- Die unsortierten assoziativen Container werden mit Hilfe von Hash-Organisationen implementiert.
- Der Standard sichert zu, dass der Aufwand für das Suchen und das Einfügen im Durchschnittsfall konstant sind, im schlimmsten Fall aber linear sein können (wenn etwa alle Objekte den gleichen Hash-Wert haben).
- Für den Schlüsseltyp muss es eine Hash-Funktion geben und einen Operator, der auf Gleichheit testet.
- Die Größe der Bucket-Tabelle wird dynamisch angepasst. Eine Umorganisation hat linearen Aufwand.

- Für die elementaren Datentypen einschließlich der Zeigertypen darauf und einigen von der Standard-Bibliothek definierten Typen wie *std*::*string* ist eine Hash-Funktion bereits definiert.
- Bei selbst definierten Schlüsseltypen muss dies nachgeholt werden. Der Typ der Hash-Funktion muss dann als dritter Template-Parameter angegeben werden und die Hash-Funktion als weiterer Parameter beim Konstruktor.
- Hierzu können aber die bereits vordefinierten Hash-Funktionen verwendet und typischerweise mit dem „$\hat{}$“-Operator verknüpft werden.

Persons.cpp

```
struct Name {
   std::string first;
   std::string last;
   Name(const std::string first, const std::string last) :
      first(first), last(last) {
   }
   bool operator==(const Name& other) const {
      return first == other.first && last == other.last;
   }
};
```

- Damit ein Datentyp als Schlüssel für eine Hash-Organisation genutzt werden kann, müssen der „==“-Operator und eine Hash-Funktion gegeben sein.

Persons.cpp

```
auto hash = [](const Name& name) {
   return std::hash<std::string>()(name.first) ^
      std::hash<std::string>()(name.last);
};
```

- *hash*<*std*::*string*>() erzeugt ein temporäres Hash-Funktionsobjekt, das einen Funktions-Operator mit einem Parameter (vom Typ *std*::*string*) anbietet, der den Hash-Wert (Typ *size_t*) liefert.
- Hash-Werte werden am besten mit dem XOR-Operator „^“ verknüpft.
- Eine Hash-Funktion muss immer den gleichen Wert für den gleichen Schlüssel liefern.
- Für zwei verschiedene Schlüssel *k1* und *k2* sollte die Wahrscheinlichkeit, dass die entsprechenden Hash-Werte gleich sind, sich 1.0 / *numeric_limits*<*size_t*>::*max*() nähern.

Persons.cpp

```
int main() {
   auto hash = [](const Name& name) { /* ... */ };
   std::unordered_map<Name, std::string, decltype(hash)> address(32,
      hash);
   address[Name("Marie", "Maier")] = "Ulm";
   address[Name("Hans", "Schmidt")] = "Neu-Ulm";
   address[Name("Heike", "Vogel")] = "Geislingen";
   std::string first; std::string last;
   while (std::cin >> first >> last) {
      auto it = address.find(Name(first, last));
      if (it != address.end()) {
         std::cout << it->second << std::endl;
      } else {
         std::cout << "Not found." << std::endl;
      }
   }
}
```

- Der erste Parameter beim Konstruktor für Hash-Organisationen legt die initiale Größe der Bucket-Tabelle fest, der zweite spezifiziert die gewünschte Hash-Funktion.

- Template-Container-Klassen benutzen implizit viele Methoden und Operatoren für ihre Argument-Typen.
- Diese ergeben sich nicht aus der Klassendeklaration, sondern erst aus der Implementierung der Template-Klassenmethoden.
- Da die implizit verwendeten Methoden und Operatoren für die bei dem Template als Argument übergebenen Klassen Voraussetzung sind, damit diese verwendet werden können, wird von Template-Abhängigkeiten gesprochen.
- Da diese recht unübersichtlich sind, erlauben Test-Templateklassen wie die nun vorzustellende *TemplateTester*-Klasse eine Analyse, welche Operatoren oder Methoden wann aufgerufen werden.

TemplateTester.hpp

```
template<class BaseType>
class TemplateTester {
   public:
      TemplateTester();
      TemplateTester(const TemplateTester& orig);
      TemplateTester(const BaseType& val);
      TemplateTester(TemplateTester&& orig);
      TemplateTester(BaseType&& val);
      ~TemplateTester();

      TemplateTester& operator=(const TemplateTester& orig);
      TemplateTester& operator=(const BaseType& val);
      TemplateTester& operator=(TemplateTester&& orig);
      TemplateTester& operator=(BaseType&& val);
      bool operator<(const TemplateTester& other) const;
      bool operator<(const BaseType& val) const;
      operator BaseType() const;

   private:
      static int instanceCounter; // gives unique ids
      int id; // id of this instance
      BaseType value;
}; // class TemplateTester
```

TemplateTester.tpp

```
template<typename BaseType>
TemplateTester<BaseType>::TemplateTester() :
      id(instanceCounter++) {
   std::cerr << "TemplateTester: CREATE #" << id <<
      " (default constructor)" << std::endl;
} // default constructor
```

- Alle Methoden und Operatoren von *TemplateTester* geben Logmeldungen auf *cerr* aus.
- Die *TemplateTester*-Klasse ist selbst eine Wrapper-Template-Klasse um *BaseType* und bietet einen Konstruktor an, der einen Wert des Basistyps akzeptiert und einen dazu passenden Konvertierungs-Operator.
- Die Klassen-Variable *instanceCounter* erlaubt die Identifikation individueller Instanzen in den Logmeldungen.

Test der *list*-Klasse

TestList.cpp

```
typedef TemplateTester<int> Test;
list<Test> myList;
// put some values into the list
for (int i = 0; i < 2; ++i) {
   myList.push_back(i);
}
// iterate through the list
for (int val: myList) {
   cout << "Found " << val << " in the list." << endl;
}
```

```
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (move constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: CREATE #3 (move constructor of 2)
TemplateTester: DELETE #2
TemplateTester: CONVERT #1 to 0
TemplateTester: CONVERT #3 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #3
clonmel$
```

TestVector.cpp

```
typedef TemplateTester<int> Test;
vector<Test> myVector(2);
// put some values into the vector
for (int i = 0; i < 2; ++i) {
   myVector[i] = i;
}
// print all values of the vector
for (int i = 0; i < 2; ++i) {
   cout << myVector[i] << endl;
}
```

```
clonmel$ TestVector >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: CREATE #1 (default constructor)
TemplateTester: ASSIGN value 0 to #0
TemplateTester: ASSIGN value 1 to #1
TemplateTester: CONVERT #0 to 0
TemplateTester: CONVERT #1 to 1
TemplateTester: DELETE #0
TemplateTester: DELETE #1
clonmel$
```

Test der *map*-Klasse

TestMap.cpp

```
typedef TemplateTester<int> Test;
map<int, Test> myMap;

// put some values into the map
for (int i = 0; i < 2; ++i) {
   myMap[i] = i;
}
```

```
clonmel$ TestMap >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: ASSIGN value 0 to #0
TemplateTester: CREATE #1 (default constructor)
TemplateTester: ASSIGN value 1 to #1
TemplateTester: CONVERT #0 to 0
TemplateTester: CONVERT #1 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #0
clonmel$
```

Test der Indizierung bei *map*

TestMapIndex.cpp

```
typedef TemplateTester<int> Test;
typedef map<Test, int> MyMap; MyMap myMap;
for (int i = 0; i < 2; ++i) myMap[i] = i;
for (const auto& pair: myMap) {
   cout << pair.second << endl;
}
```

```
clonmel$ TestMapIndex >/dev/null
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (move constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: COMPARE #1 with #2
TemplateTester: CREATE #3 (move constructor of 2)
TemplateTester: COMPARE #1 with #3
TemplateTester: COMPARE #3 with #1
TemplateTester: DELETE #2
TemplateTester: DELETE #3
TemplateTester: DELETE #1
clonmel$
```

- Die Algorithmen der STL sind über **#include** <algorithm> zugänglich.
- Die Algorithmen arbeiten alle auf Sequenzen, die mit Iteratoren spezifiziert werden.
- Sie unterteilen sich in
  - nicht-modifizierende Algorithmen auf Sequenzen
  - Algorithmen, die Sequenzen verändern und
  - weitere Algorithmen, wie Sortieren, binäre Suche, Mengen-Operationen, Heap-Operationen und die Erzeugung von Permutationen.

Der Standard legt folgende Komplexitäten fest. Hierbei ist n normalerweise die Länge der Sequenz.

$O(1)$	*swap*(), *iter_swap*()
$O(\log n)$	*lower_bound*(), *upper_bound*(), *equal_range*(), *binary_search*(), *push_heap*(), *pop_heap*()
$O(n \log n)$	*inplace_merge*(), *stable_partition*(), *sort*(), *stable_sort*(), *partial_sort*(), *partial_sort_copy*(), *sort_heap*()
$O(n^2)$	*find_end*(), *find_first_of*(), *search*(), *search_n*()
$O(n)$	alle anderen Funktionen

(Siehe Abschnitt 25 im Standard und 32.3.1 bei Stroustrup.)