



**C++**

**ChaosSeminar**

**Chaos Computer Club Ulm**

**Alexander Bernauer**

**[alexander.bernauer@ulm.ccc.de](mailto:alexander.bernauer@ulm.ccc.de)**

# Was ist C++?

- **Bjarne Stroustrup implementiert einen Simulator**
  - **Simular**  $\Rightarrow$  zu langsam
  - **BCPL**  $\Rightarrow$  zu aufwendig
- **Erweiterung von C um Klassen**
- **Entwicklung vom Tool zur eigenen Sprache**
- **Cfront C++ Compiler**
- **Cfront Release 3.0**
- **ISO C++**

# Designziele

- Effizienz
- Modellierungstechniken
- Verwendbarkeit
- Verfügbarkeit
- Pragmatismus

# Warum C?

- C ist flexibel
- C ist effizient
- C ist verfügbar
- C ist portabel

# Nachteile der Kompatibilität

- Syntax
- Präprozessor
- Linkage Model
- wenig Schlüsselworte

## **weitere Einflüsse**

- **Simula**
  - **Klassen**
- **Algol68**
  - **Operatoren überladen**
  - **Referenzen**
  - **freie Deklaration**
- **Betriebssysteme**
  - **Zugriffskontroller**
  - **const**
- **BCPL**
  - **// Kommentare**

# Skipping

- **C Subset**
  - **C-Syntax**
  - **Schleifen, Bedingungen, ...**
  - **Präprozessor**
  - **Zeiger**
  - **Funktionen**
  - **Structs**
  - **Enums**

# Klassen

## vector.h

```
1 class Vector {
2 public:
3     Vector(int size);    // Konstruktor
4     ~Vector();          // Destruktor
5
6     int getSize();
7     // ...
8 private:
9     int _size;
10    int* _array;
11 };
```



# Klassen

## vector.cpp

```
1 Vector::Vector(int size)
2     : _size(size)
3 {
4     _array = new int[_size];
5 }
6
7 Vector::~~Vector()
8 {
9     delete [] _array;
10 }
11
12 int Vector::getSize()
13 {
14     return _size;
15 }
```

# Klassen

## main

```
1 Vector v(4);  
2 Vector w = Vector(5);  
3  
4 v.getSize();
```

# Funktionen überladen

## vector.h

```
1 class Vector {
2 public:
3     Vector(int size);
4     Vector();
5     Vector(int size, int value);
6     //...
7 };
```

# Funktionen überladen

## vector.cpp

```
1 Vector::Vector()
2     : _size(0)
3 {
4     _array = 0;
5 }
6
7 Vector::Vector(int size, int value)
8     : _size(size)
9 {
10    _array = new int[_size];
11    for (int i=0; i<_size; i++) _array[i] = value;
12 }
13
14 Vector::~~Vector()
15 {
16     if (_array) delete [] _array;
17 }
```

# Defaultwerte

## vector.h

```
1 class Vector {
2 public:
3     // Vector(int size);
4     // Vector();
5
6     Vector(int size=0);
7     //...
8 };
```

# Defaultwerte

## vector.cpp

```
1 // Vector::Vector() { ... }
2
3 // Vector::Vector(int size) { ... }
4
5 Vector::Vector(int size)
6     : _size(size)
7 {
8     if (_size) {
9         _array = new int[_size];
10    } else {
11        _array = 0;
12    }
13 }
```

# Defaultwerte

main

```
1 Vector v; // Vector v(0);
```

# operator[]

## vector.h

```
1 class Vector {  
2 public:  
3 // ...  
4     int operator[](int index);  
5 };
```



# operator[]

## vector.cpp

```
1 int Vector::operator[](int index)
2 {
3     if (index >= _size) {
4         {
5             /* Fehler behandeln. z.B. Exception werfen */
6         }
7
8     return _array[index];
9 }
```

# operator[]

## main

```
1 Vector v(4, 1);  
2  
3 int i = v[3];
```

# Exceptions

## vector.h

```
1 class OutOfBounds {
2 public:
3     OutOfBounds(int maxIndex);
4     int getMaxIndex();
5 private:
6     int _maxIndex;
7 }
8
9 class Vector {
10 public:
11     //...
12     int operator[](int index) throw (OutOfBounds);
13 };
```

# Exceptions

## vector.cpp

```
1 OutOfBounds::OutOfBounds(int maxIndex)
2     : _maxIndex(maxIndex)
3 {}
4
5 int OutOfBounds::getMaxIndex()
6 {
7     return _maxIndex;
8 }
9
10 int Vector::operator[](int index)
11 {
12     if (index >= _size) {
13         {
14             throw OutOfBounds(_size)
15         }
16
17     return _array[index];
18 }
```

# Exceptions

## main

```
1 Vector v(2);  
2  
3 try {  
4     v[5];  
5 } catch (OutOfBounds e) {  
6     // Fehler und e.getMaxIndex() ausgeben  
7 }
```

## **inline**

- **Hinweis an den Compiler, Funktionsaufruf einzusparen**
- **Zentrale Technik zur Vereinigung von Abstraktion und Effizienz**
- **Funktionsdefinition muss zur Compilezeit vorhanden sein**
- **Bedachter Einsatz wichtig**

# inline

## vector.h - Variante 1

```
1 class Vector {
2 public:
3     // ...
4     inline int getSize();
5     // ...
6 private:
7     int _size;
8     int* _array;
9 };
10
11 int Vector::getSize()
12 {
13     return _size;
14 }
```

# inline

## vector.h - Variante 2

```
1 class Vector {
2 public:
3     // ...
4     int getSize()
5     {
6         return _size;
7     }
8     // ...
9 private:
10    int _size;
11    int* _array;
12 };
```



# inline

## main

```
1 inline int sum(Vector* v)
2 {
3     int sum = 0;
4     for (int i = 0; i < v->getSize(); i++) { // getSize ist normal kein Funktionsaufruf
5         sum += (*v)[i]; // das ist hässlich. Mit Referenzen ist es besser
6     }
7     return sum;
8 }
9
10
11 Vector v;
12 int summe = sum(&v); // sum ist wahrscheinlich kein Funktionsaufruf
```

# const vector.h

```
1 class Vector {
2 public:
3     //...
4     void setValue(int index, int value);
5
6     // int getSize();
7     int getSize() const;
8     // ...
9 };
```

# const

## vector.cpp

```
1 void setValue(int index, int value)
2 {
3     if (index >= _size) {
4         throw OutOfBounds(index);
5     }
6
7     _array[index] = value;
8 }
9
10 int Vector::getSize() const
11 {
12     return _size;
13 }
14
15
```

# const

## main

```
1 const Vector v(4);  
2  
3 v.setValue(2, 9); // error: const qualifier discarded  
4  
5 int i = v[3]; // ok: read-only-Zugriff
```

# Referenzen

## main

```
1 Vector v(3);  
2  
3 v.setValue(1, 2); // verhält sich nicht wie built-in Array  
4  
5 v[1] = 2; // error: non-lvalue in assignment
```

# Referenzen

- wahlweise
  - konstanter Zeiger mit automatischer Dereferenzierung
  - alias Name
- Call-by-Reference ohne Zeiger
- Zugriff auf interne Daten ohne Zeiger

# Referenzen

## example.cpp

```
1 int i;
2 int& r;      // Fehler: Referenz muss initialisiert werden
3 int& r = i;  // Referenz initialisieren
4 r = 3;      // i hat den Wert 3
5
6
7 int* p = &i;
8 int& r2 = *p; // Zeiger in Referenz umwandeln
9
10
11 void inc(int& i) { i++; }
12
13 int j = 4;
14 inc(j);      // call-by-reference
15 // j = 5
16
17
18 void bar(const int& i) {}
19 bar(3);      // temporäre Variable
```

# Referenzen

## vector.h

```
1 class Vector {
2 public:
3 // ...
4     // void setValue(int index, int value)
5     // int operator[](int index);
6
7     int& operator[](int index);
8 };
```



# Referenzen

## vector.cpp

```
1 int& Vector::operator[](int index)
2 {
3     if (index >= _size) {
4         {
5             throw OutOfBounds(_size)
6         }
7
8     return _array[index];
9 }
```

# Referenzen

## main

```
1 Vector v(3);
2
3 v[1] = 2; // ok
4
5
6 void sum(const Vector& v); // Garantie, dass v nicht verändert wird
7
8 sum(v); // call-by-reference
9
10
11 void sum(const Vector& v)
12 {
13     int sum = 0;
14     for (int i=0; i<v.getSize(); i++) {
15         sum += v[i]; // Fehler: nicht-const Funktionsaufruf auf const Objekt
16     }
17 }
```

# Referenzen

## vector.h

```
1 class Vector {  
2 public:  
3 // ...  
4     int& operator[](int index);  
5     const int& operator[](int index) const;  
6 };
```

# Copy-Konstruktor

- Kopieren wird vom copy Konstruktor gemacht
- der Default Copy Konstruktor macht eine byteweise Kopie

# Copy-Konstruktor

## vector.h

```
1 class Vector {
2 public:
3     //...
4     Vector(const Vector& v); // copy Konstruktor
5     //...
6 };
```

# Copy-Konstruktor

## vector.cpp

```
1 Vector::Vector(const Vector& v)
2     : _size(v._size)
3 {
4     _array = new int[_size];
5     memcpy(_array, v._array, _size * sizeof(int));
6 }
7
```

# Copy-Konstruktor

## main

```
1 Vector v;  
2 Vector w1(v); // Initialisierung durch copy Konstruktor  
3 Vector w2 = v; // Initialisierung durch copy Konstruktor  
4  
5 void foo(Vector w) {  
6     // w ist Kopie von v  
7 }  
8  
9 foo(v);  
10
```

## **operator=**

- **Zuweisung wird durch den operator= erledigt**
- **Default operator= macht eine byteweise Kopie**



# operator= vector.h

```
1 class Vector {  
2 public:  
3     //...  
4     Vector& operator=(const Vector& v);  
5 };
```

# operator=

## vector.cpp

```
1 Vector& operator=(const Vector& v)
2 {
3     _size = v._size;
4     _array = new int[_size];
5     memcpy(_array, v._array, _size * sizeof(int));
6
7     return *this;
8 }
```

# operator=

## example

```
1 Vector v;  
2 Vector w = v; // Initialisierung => copy Konstruktor  
3 Vector y, z;  
4  
5 z = v; // Zuweisung => operator=  
6  
7 z = y = v; // Kaskadierung von Zuweisungen  
8  
9 Vector Foo(Vector v) {  
10     // copy Konstruktor zur Initialisierung des Parameters  
11     return v;  
12     // copy Konstruktor zur Initialisierung des Rückgabewertes  
13 }  
14  
15 Vector a;  
16 a = Foo(z); // insgesamt zweimal copy Konstruktor und ein mal operator=  
17  
18  
19 Vector b = Foo(z); // nur zweimal copy Konstruktor
```

# Templates

- **Vector Klasse ist Container für ints**
- **um Container für doubles zu bekommen**
  - **von Hand kopieren**
  - **int durch double austauschen**
- **Problem, wenn man beliebige Typen unterstützen will (z.B. als Bibliothek)**
- **Templates bieten dafür eine generische Lösung**

# Templates

## vector.h

```
1 template <class T>
2 class Vector {
3     Vector(int size=0)
4         : _size(size)
5     {
6         if (_size) _array = new T[_size];
7         else array = 0;
8     }
9
10    // ...
11
12    T& operator[](int index)
13    {
14        // Bereichsprüfung
15        return _array[index];
16    }
17
18    // ...
19 private:
20    T* _array;
21    int _size;
22 };
```

# Templates

## main

```
1 Vector<int> v;
2
3 typedef Vector<double> DV;
4
5 DV w;
6
7
8 template <class T> T sum(const Vector<T>& v)
9 {
10     T sum = 0; // T(int) muss definiert sein
11     for (int i=0; i < v.getSize(); i++) {
12         sum += v[i]; // operator += muss für T definiert sein
13     }
14     return sum;
15 }
16
17
18 int s = sum< Vector<int> >(v); // explizite Angabe der Templateparameter
19
20 double d = sum(w);           // Compiler ermittelt Templateparameter
```

# Templates

- gängige Compiler und Linker unterstützen Templates nicht gut
- zum Zeitpunkt der Verwendung will der Compiler den Code erzeugen
- Code muss im Headerfile stehen
- der Linker müsste nachträglich den Compiler aufrufen können

# Templates

- **Konstanten als Templateparameter**
- **eigene Implementierung für einen Typ anbieten (Spezialisierung)**
- **Defaultwerte für Templateparameter**
- **Membertemplates**



# Ersatz für den Präprozessor

## C-Style

```
1 #define min(x,y) (((x) < (y)) ? (x) : (y))
2
3 int foo();
4 int x;
5
6 int main()
7 {
8     min(foo(), 3); // foo wird zwei mal gerufen, falls foo() < 3
9
10    min(x++, 4); // x wird zweimal inkrementiert, falls x < 3
11 }
```

# Ersatz für den Präprozessor

## C++-Style

```
1 template <class T>
2 inline const T& min(const T& a, const T& b)
3 {
4     return (a < b) ? a : b;
5 }
6
7 int foo();
8 int x;
9
10 int main()
11 {
12     min(foo(), 3); // foo() wird einmal gerufen. Temporäre Variable wird erstellt
13
14     min(x++, 4); // x wird einmal inkrementiert, nachdem min zurück kehrt
15 }
```

# Vererbung

## dynvector.h

```
1 template <class T>
2 class DynVector : public Vector<T> {
3 public:
4     DynVector(int size=0) : Vector<T>(size) {}
5     DynVector(int size, T& value) : Vector<T>(size, value) {}
6
7     T& operator[](int index)
8     {
9         if (index >= _size) {
10             T *const oldArray = _array;
11             const int oldSize = _size;
12
13             while(_size <= index) _size *= 2; // nicht im Sonderfall _size == 0
14
15             _array = new T[_size];
16             for (int i = 0; i < oldSize; i++ ) {
17                 _array[i] = oldArray[i];
18             }
19
20             delete [] oldArray;
21         }
22         return _array[index];
23     }
24 };
```

# Vererbung

## vector.h

```
1 template <class T>
2 class Vector {
3 //...
4
5 // private:
6 //     T* _array;
7 //     int _size;
8 protected:
9     T* _array;
10    int _size;
11 };
```

# Vererbung

## main

```
1 DynVector<int> v(1);  
2  
3 for (int i=0; i<10; i++) {  
4     v[i] = i;  
5 }
```

# virtuelle Funktionen

## main

```
1 void fill(Vector<int>& v, int count, int value)
2 {
3     for (int i = 0; i < count; i++) {
4         v[i] = value; // Vector<T>::operator[]() wird gerufen.
5                       // Kann fehlschlagen, falls count > v.getSize()
6     }
7 }
8
9 DynVector<int> v(1);
10
11 fill(v, 4, 3);
```

## **virtuelle Funktionen**

- immer "die richtige" Funktion rufen
- Hinzufügen von Typinformationen
- Entscheidung zur Laufzeit
- übliche Implementierung mittels vtable
- eine Klasse mit mindestens einer virtuellen Funktion heißt **polymorph**

# virtuelle Funktionen

## vector.h

```
1 template <class T> class Vector {  
2 public:  
3     virtual T& operator[](int index) { /*...*/ }  
4 };
```



# virtuelle Funktionen

## main

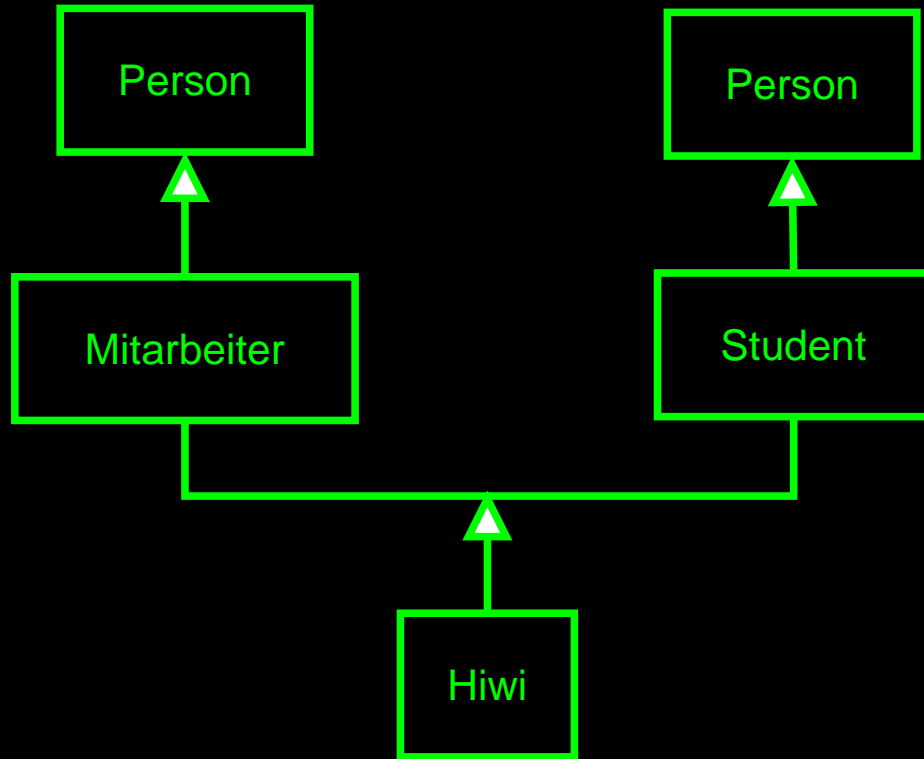
```
1 void fill(Vector<int>& v, int count, int value)
2 {
3     for (int i = 0; i < count; i++) {
4         v[i] = value; // DynVector<T>::operator[] wird gerufen,
5                       // falls v vom Typ DynVector ist
6     }
7 }
8
9 DynVector<int> v(1);
10
11 fill(v, 4, 3);
```

# Mehrfachvererbung

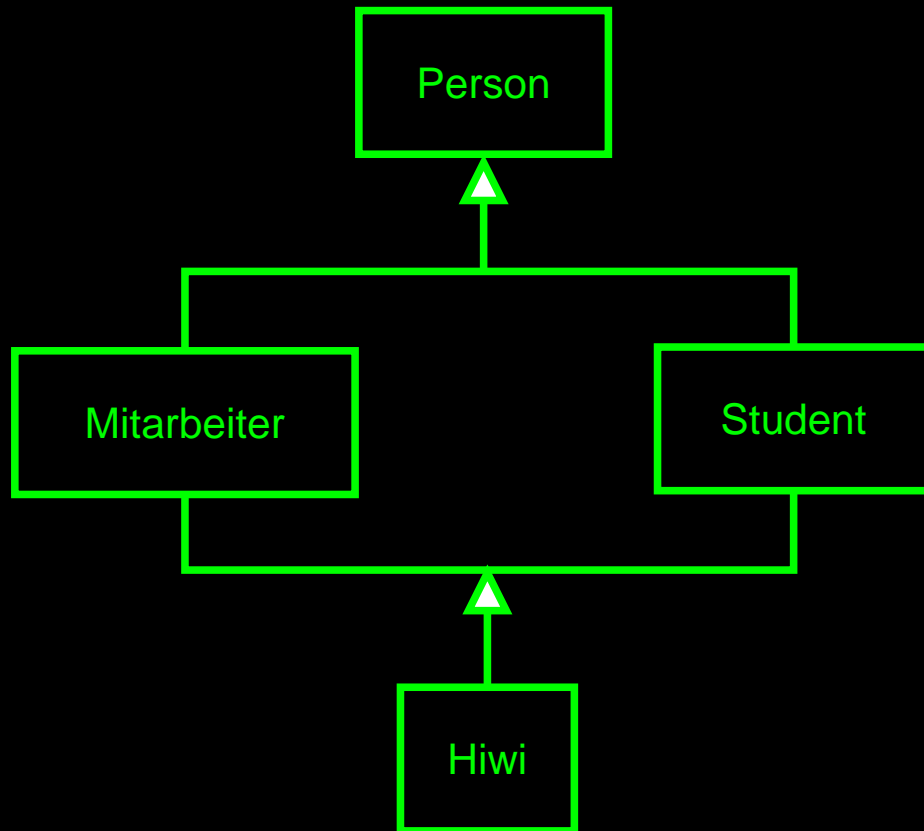
## example

```
1 class Person { /* Name, Adresse, ... */ };
2
3 class Student : public Person { /* Matrikel Nummer, Studiengang, ... */ };
4 class Mitarbeiter : public Person { /* Personal Nummer, Abteilung, ... */ };
5
6 class Hiwi : public Student, public Mitarbeiter { /* Bewertung */ };
```

# Mehrfachvererbung



# Mehrfachvererbung



# Mehrfachvererbung

## example

```
1 class Person { /* Name, Adresse, ... */ };
2
3 class Student : virtual public Person { /* Matrikel Nummer, Studiengang, ... */ };
4 class Mitarbeiter : virtual public Person { /* Personal Nummer, Abteilung, ... */ };
5
6 class Hiwi : public Student, public Mitarbeiter { /* Bewertung */ };
```

# abstrakte Klassen

## interface.h

```
1 class Interface {
2 public:
3     virtual void process() =0; // abstrakte Funktion
4     //...
5 };
6
7 Interface& factory();
8
```

# abstrakte Klassen

## implementation.cpp

```
1 #include "interface.h"
2
3 class Implementation : public Interface {
4 public:
5     void process() { /*...*/ }
6 };
7
8 Interface& factory()
9 {
10     return *new Implementation();
11 }
```

# abstrakte Klassen

## main

```
1 #include "interface.h"
2
3 int main()
4 {
5     Interface& o = factory();
6
7     o.process();
8 }
```



# Standard Template Library

- **libstdc++ seit ISO C++**
- **Inhalt**
  - **IO-Streams**
  - **Strings**
  - **Prädikate**
  - **Container**
  - **Iteratoren**
  - **Algorithmen**

# Strings

## example

```
1 void output(const string&);
2
3 int main()
4 {
5     string s = "Hallo";
6     string t = s;
7     t = t + " ";
8     t += "Welt";
9
10    output(t);
11
12    t[0] = 'h';
13
14    output(t);
15 }
16
17 void output(const string& s)
18 {
19     printf("ausgabe: %s\n", s.c_str());
20 }
```

# Strings - weitere Möglichkeiten

- vergleichen
- einfügen
- suchen
- ersetzen
- Teilstrings extrahieren
- Verwendung in STL Algorithmen

# **Funktoren - operator()**

- **Kapselung von Algorithmen in Klassen**
- **Definition des operator()**
- **Übergabe eines Objektes der Funktorklasse**
- **Verwendung des Funktors wie eine Funktion**

# Prädikate

- spezielle Funktoren mit Rückgabewert bool
- Parametrisierung von STL Algorithmen

# Prädikate

## less.h

```
1 template <class T>
2 class less {
3 public:
4     less(T t)
5         : _rhs(t)
6     {}
7
8     bool operator()(T lhs)
9     {
10         return lhs < _rhs;
11     }
12
13 private:
14     T _rhs;
15 };
```

# Prädikate

## main

```
1 #include "less.h"
2
3 template <class T> int find(const Vector<int>& v, T predicate)
4 {
5     for (int i=0; i < v.getSize(); i++) {
6         if (predicate(v[i])) {
7             return v[i];
8         }
9     }
10    // Sonderfallbehandlung
11 }
12
13 template <class T> less<T> less_than(T t)
14 {
15     return less<T>(t);
16 }
17
18 int main()
19 {
20     Vector<int> v;
21     // v mit ints füllen
22
23     int i = find(v, less<int>(3));
24     int i = find(v, less_than(3));
25 }
```

# Container

- **einfache Container**
  - **vector**
  - **list**
  - **dequeue**
- **Container Adapter**
  - **stack**
  - **queue**
  - **priority\_queue**
- **assoziative Container**
  - **map**
  - **multimap**
  - **set**
  - **multiset**



# Container

## priority\_queue

```
1 struct Packet {
2     int seq;
3     // ...
4 };
5
6 class Compare {
7 public:
8     bool operator()(const Packet& lhs, const Packet& rhs) { return lhs.seq < rhs.seq; }
9 };
10
11 std::priority_queue<Packet, deque<Packet>, Compare> q;
12
13 void readPackages() {
14     while ( /* packets available */ ) {
15         // read packet from network
16         q.push(packet);
17     }
18 }
19
20 void process() {
21     while (! q.empty()) {
22         // process q.top()
23         q.pop();
24     }
25 }
```

# Container

## map

```
1 struct Compare
2 {
3     bool operator()(const char* s1, const char* s2) const
4     {
5         return strcmp(s1, s2) < 0;
6     }
7 };
8
9 typedef std::map<const char*, int, Compare> Map;
10
11 Map m;
12
13 m["zwei"] = 2;
14 m["eins"] = 1;
15
16 Map::iterator iter = m.find("eins");
17 std::pair<const char*, int> result = *iter;
18
19 string key = result.first;
20 int value = result.second;
```

# Iteratoren

- **Abstraktion zur Abarbeitung einer Sequenz**
- **Alles, was sich wie ein Iterator verhält, ist ein Iterator**
- **Container, Strings und Streams haben Iteratoren**
- **Zentrale Rolle für STL Algorithmen**

# Iteratoren - Eigenschaften

- Dereferenzierung `operator*` und `operator->`
- nächstes Element `operator++`
- Vergleich `operator==`

# Iteratoren

## example

```
1 void process(const vector<string>& v)
2 {
3     vector<string>::iterator iter;
4     for (iter = v.begin(); iter != v.end(); v++) {
5         string akt = *iter;
6         // akt verarbeiten
7     }
8 }
```

# Algorithmen - Teil 1

- **nichtmodifizierende Sequenzoperationen**

- **for\_each**
- **find**
- **count**
- **equal**
- **search**

- **modifizierende Sequenzoperationen**

- **copy**
- **transform**
- **replace**
- **fill**
- **generate**
- **remove**
- **unique**
- **reverse**
- **random\_shuffle**

# Algorithmen - Teil 2

- **sortierte Sequenzen**

- **sort**
- **binary\_search**
- **merge**
- **partition**

- **Mengenalgorithmen**

- **includes**
- **set\_union**
- **set\_intersection**
- **set\_difference**

- ...

# Algorithmen

## Implementierung

```
1 template <class In, class Op> Op for_each(In begin, In end, Op f)
2 {
3     while (begin != end) f(*begin++);
4     return f;
5 }
6
7 template <class In, class T> In find(In begin, In end, const T& value)
8 {
9     while (begin != end) {
10         if (*begin == value) break;
11         ++begin;
12     }
13     return begin;
14 }
15
16 template <class In, class Out> Out copy(In begin, In end, Out out)
17 {
18     while (begin != end) *out++ = *begin++;
19     return out;
20 }
```



# Algorithmen

## main

```
1 int gen() {
2     static int count = 0;
3     return count++;
4 }
5
6 class Mod {
7 public:
8     Mod(int m) : m(m) {};
9     int operator()(int i) { return i % m; }
10 private:
11     int m;
12 };
13
14 int array[20];
15 vector<int> v(20);
16
17 int main()
18 {
19     generate(array, &array[20], gen);
20     transform(array, &array[20], v.begin(), Mod(3));
21     sort(v.begin(), v.end());
22     copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
23 }
```

# Möglichkeiten der Speicherverwaltung

- **statische Variablen**
  - Speicher kann nicht freigegeben werden
- **automatische Variablen auf dem Stack**
  - werden beim Verlassen des Blocks zerstört
- **dynamische Variablen auf dem Heap**
  - müssen von Hand zerstört werden
- **automatische Speicherverwaltung für dynamische Variablen**
  - Garbagecollection?

# Smartpointer

## main

```
1 #include "smartptr.h"
2
3 class Data {
4 public:
5     void foobar();
6     /* irgendwelche Daten */
7 };
8
9 typedef SmartPtr<Data> SP;
10
11 SP newData() {
12     Data* d = new Data;
13     // ....
14     return SP(d);
15 }
16
17 void processData(const Data& d);
18
19 void worker() {
20     while (1) {
21         SP p = newData();
22         p->foobar();
23         processData(*p);
24     }
25 }
```

# Smartpointer

## reference.h

```
1 template <class T>
2 class Reference {
3 public:
4     Reference(T* data) : _refCounter(1), _data(data) { }
5     ~Reference() { delete _data; }
6     void aquire() { _refCounter++; }
7     bool release()
8     {
9         _refCounter--;
10        return (_refCounter == 0);
11    }
12
13    T* getData()
14    {
15        return _data;
16    }
17
18 private:
19     int _refCounter;
20     T* _data;
21 };
```

# Smartpointer

## smartpointer.h

```
1 #include "reference.h"
2
3 template <class T>
4 class SmartPtr {
5 public:
6     SmartPtr() : _reference(0) { }
7
8     SmartPtr(T* data) : _reference(new Reference<T>(data)) { }
9
10    SmartPtr(const SmartPtr& p) : _reference(p._reference) { _acquire(); }
11
12    ~SmartPtr() { _release(); }
13
14    SmartPtr& operator=(const SmartPtr& p)
15    {
16        if (this == &p) return;
17        _release();
18        _reference = p._reference;
19        _acquire();
20    }
21
22    T& operator*() { return *(_reference->getData()); }
23
24    T* operator->() { return _reference->getData(); }
25
```

```
31 private:
32     void _release() {
33         if (_reference) {
34             if (_reference->release()) {
35                 delete _reference;
36             }
37         }
38     }
39
40     void _acquire() {
41         if (_reference) {
42             _reference->aquire();
43         }
44     }
45
46     Reference<T>* _reference;
47 };
48
```

## **missing**

- **Namespaces**
- **IO-Streams**
- **operator new**
- **Allokatoren**
- **pointer to memberfunction**
- **Runtime Typeinformation (RTTI)**
- **SGI-Extensions**
- **Boost Library**

## verwendete Software

- vim <http://www.vim.org/>
- latex <http://www.latex-project.org/>
- texify
- acroread
- Debian GNU/Linux