Arrays in STL (array)

Objekte vom Datentyp `array<`T`, `n`>` speichern n viele Elemente vom Typ T. Es handelt sich im Wesentlichen um dünne wrapper um eingebaute C-Arrays vom Typ T`[`n`]`. Der Vorteil gegenüber der Verwendung von C-Arrays liegt in der, im Gegensatz zu C-Arrays, mit anderen container-Datentypen aus der STL einheitlichen Programmierschnittstelle in Form der angebotenen Methoden und Attribute.

Im Folgenden bezeichnet T einen Datentyp, n einen „hinreichend konstanten“ Ausdruck vom Typ `size_t`, $k < n$ einen Wert vom Typ `size_t`, $t_0, \ldots, t_{n-1}$ Werte vom Typ T und a, a' Variablen vom Typ `array<`T`, `n`>` Zudem bezeichnet i einen Iterator über a.

`array<`T`, `n`> `a	Vereinbart Variable a vom Typ `array<`T`, `n`>` und allen Elementen initialisiert mit Standardkonstruktor
`array<`T`, `n`> `a{t_0`, ...,` t_{k-1}`}`	Vereinbart Variable a vom Typ `array<`T`, `n`>` mit gegebenen Präfix von Elementen und allen verbleibenden Elementen initialisiert mit Standardkonstruktor
a`.at(`k`)`	Liefert Referenz auf Element mit Index k, wirft exception falls k kein valider Index
a`[`k`]`	Liefert Referenz auf Element mit Index k, undefiniertes Verhalten falls k kein valider Index
a`.front()` a`.back()`	Liefert Referenz auf erstes bzw. letztes Element von a falls vorhanden, sonst undefiniertes Verhalten
a `==` a' a `!=` a' a `>` a' a `<` a' a `<=` a' a `>=` a'	Vergleicht a und a' lexikographisch
a`.size()`	Liefert n
a`.empty()`	Liefert `true` g.d.w. a leer ist (d.h. $n = 0$)
`array<`T`, `n`>::iterator` i `array<`T`, `n`>::const_iterator` i `array<`T`, `n`>::reverse_iterator` i `array<`T`, `n`>::const_reverse_iterator` i	Vereinbart Variable i als Vorwärts-, Konstanten-, Rückwärts- bzw. Rückwärts-Konstanten-Iterator
a`.begin()` a`.end()` a`.rbegin()` a`.rend()`	Liefert Vorwärts-Iterator über a zum ersten bzw. *nach* dem letzten Element bzw. Rückwarts-Iterator über a zum letzten
`++`i i`++` `--`i i`--`	Verschiebt Iterator i auf nächste bzw. vorherige Position in Durchlaufrichtung
i `+` k k `+` i i `-` k	Liefert Iterator i verschoben um k in Durchlaufrichtung
i `+=` k i `-=` k	Verschiebt Iterator i um k in Durchlaufrichtung
`*`i	Referenz zu Wert an aktueller Position von i
i`[`k`]`	`*(`i `+` k`)`
i`->`c	`*(`i`).`c

Beispiel. Wir initialisieren und sortieren einen `array` der Länge drei mit Elementen vom Typ `double`.

array_usage.cpp

```cpp
#include <iostream>
#include <array>
#include <algorithm>

using namespace std;

int main() {
  array<double, 3> a1{3, 2, 1};

  sort(a1.begin(), a1.end());

  cout << "a1[0.."
       << a1.size() - 1
       << "]:";
  for (const double& x: a1)
    cout << " " << x;
  cout << endl;
}
```

```
a1[0..2]: 1 2 3
```

Template Deklarationen

⟨Templ'Dekl'⟩ ⟶ “`template`” “<” [⟨Templ'Param's⟩] “>” ⟨Dekl'⟩
⟨Templ'Param's⟩ ⟶ ⟨Templ'Param'⟩ {“,” ⟨Templ'Param'⟩}
⟨Templ'Param'⟩ ⟶ ⟨Templ'Typ'Param⟩ | ⟨Param'Dekl'⟩
⟨Templ'Typ'Param'⟩ ⟶ (“`class`” | “`typename`”) [⟨Ident⟩] [“=” ⟨Typ⟩]
| “`template`” “<” [⟨Templ'Param's⟩] “>” “`class`” [⟨Ident⟩] [“=” ⟨Id'Expr'⟩]

Das Schlüsselwort `template` deklariert eine parametrisierte, nicht-abgeschlossene Familie von „Objekten“. An Stelle von ⟨Dekl'⟩ kann die Deklaration einer Funktion, eines Typaliases, einer Variable oder einer Klasse stehen. Die Familie ist parametrisiert anhand der in ⟨Templ'Param's⟩ angegebenen Parameter. Die einzelnen Parameter ⟨Param'Dekl'⟩ sind hierbei entweder von der selben Form wie Funktionsparameter, ⟨Param'Dekl'⟩ oder Typparameter ⟨Templ'TypParam'⟩. Als Typparameter kann auch wiederum ein template mit Parametern gefordert werden.

Ob in ⟨Templ'Typ'Param'⟩ `class` oder `typename` verwendet wird macht keinen Unterschied. `class` ist jedoch aus Konsistenzgründen vorzuziehen.

Instanziierung von Templates

⟨Templ'Inst'⟩ ⟶ ⟨Templ'Name⟩ “<” [⟨Templ'Arg'⟩ {“,” ⟨Templ'Arg'⟩}] “>”
⟨Templ'Arg'⟩ ⟶ ⟨Expr⟩ | ⟨Type⟩

Bevor ein konkretes „Objekt“ aus einer mit `template` eingeführten Familie ⟨Templ'Inst'⟩ verwendet werden kann, muss es *instanziiert* werden. Es werden hierbei konkrete Argumente für die Parameter eingesetzt und

alle Erwähnungen der Parameter in der Deklaration entsprechend ersetzt.

⟨Expl'Templ'⟩ → "`template`" ⟨Inst'Dekl'⟩

Templates können vor Ihrer Verwendung explizit instanziiert werden. Hierfür wird ⟨Expl'Templ'⟩ der Deklaration des Objektes das Schlüsselwort `template` vorgestellt. ⟨Inst'Dekl'⟩ darf hierbei keine Definition enthalten und es wird für den Namen des „Objekts" ⟨Templ'Inst'⟩ eingesetzt.

Beispiel. Wir instanziieren ein Mitglied des Templates `array` explizit.

instantiate_array_demo.cpp

```cpp
#include <array>

using namespace std;

template class std::array<double, 3>;

int main() {
  return 0;
}
```

I.d.R. wird auf die explizite Instanziierung von Templates verzichtet. Bei erster Erwähnung einer Instanz ⟨Templ'Inst'⟩ wird das template automatisch passend instanziiert.

Beispiel. Wir vereinbaren ein Funktionstemplate für p-Normen auf `array`s beliebiger Länge. Im Hauptprogramm verwenden wir das Funktionstemplate. Es wird für jeden Satz von Parametern jeweils einmalig eine Funktion instanziiert.

array_pnorm.cpp

```cpp
#include <iostream>
#include <array>
#include <cstddef>
#include <cmath>

using namespace std;

template<class T = double, unsigned int p = 2, size_t dim>
  T pNorm(const array<T, dim>& v) {
    T x = 0;
    for (const T& y: v)
      x += pow(y, p);
    return pow(x, static_cast<T>(1) / p);
}

int main() {
  array<double, 3> a1{1, 2, 3};
  cout << pNorm<double, 2, 3>(a1) << endl;

  array<double, 5> a2{5, 4, 3, 2, 1};
  cout << pNorm<double, 3>(a2) << endl;
}
```

```
3.74166
6.0822
```

Beispiel. Wir vereinbaren ein Variablentemplate `pi_v` für Darstellungen der Kreiszahl π als Wert eines beliebigen Typs in den sich ein Wert vom Typ `long double` implizit konvertieren lässt.

template_pi.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <limits>

using namespace std;

template<class T>
  constexpr T pi_v = static_cast<T>(3.14159265358979323846264338327950288 4L);

int main() {
  cout
    << pi_v<int> << endl
    << setprecision(numeric_limits<float>::digits10 + 1)
    << pi_v<float> << endl
    << setprecision(numeric_limits<double>::digits10 + 1)
    << pi_v<double> << endl
    << setprecision(numeric_limits<long double>::digits10 + 1)
    << pi_v<long double> << endl;
}
```

```
3
3.141593
3.141592653589793
3.141592653589793239
```

Beispiel. Wir vereinbaren einen parametrisierten Typalias `downset<T>` für Mengen von Werten eines beliebigen Typs T. Wir übergeben hierbei an `set` aus STL einen weiteren Template-Parameter um zu erreichen, dass die Werte in umgekehrter Reihenfolge gespeichert werden.

template_downset.cpp

```cpp
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template<class Key>
  using downset = set<Key, greater<Key>>;

int main() {
  downset<int> s;
  s.insert(3); s.insert(1); s.insert(2);
```

```
  cout << "s:";
  for (const int& x: s)
    cout << " " << x;
  cout << endl;
}
```

```
s: 3 2 1
```

Beispiel. Wir vereinbaren einen parametrisierten Typalias `ptr<T>` für den Typ „Zeiger auf T“.

template_ptr.cpp

```
#include <iostream>

using namespace std;

template<class T>
  using ptr = T*;

ptr<int> x = nullptr;

int main() {
  int y = 7;
  x = &y;

  cout << *x << endl;
}
```

```
7
```

Beispiel. Wir implementieren eine eigene Klasse `Vektor` in analogie zum STL-Datentyp `vector`. Wie `vector` ist `Vektor` ein über den Elementtyp parametrisiertes Template.

poly_vector.cpp

```
#include <iostream>

using namespace std;

template<class T>
  class Vektor {
    private:
      T* ap;
      int len;

    public:
      Vektor(int n = 0, const T& x = T{}): ap(nullptr), len(n) {
        if (!len) return;
        ap = new T[n];
        for (int i = 0; i < len; i++)
          ap[i] = T{x};
      }
```

```
    ~Vektor() { if (ap) delete[] ap; }
    Vektor& operator=(const Vektor& b) = delete;

    class iterator {
      friend class Vektor;

      private:
        Vektor* v;
        int pos;

        iterator(Vektor* v_, int pos_ = 0)
          : v(v_), pos(pos_) {}

      public:
        T& operator*() { return v->ap[pos]; }
        iterator& operator++() {
          if (pos < v->len) pos++;
          return *this;
        }
        bool operator!=(const iterator& other) const {
          return v != other.v || pos != other.pos;
        }
    };

    iterator begin() { return iterator{this}; }
    iterator end() { return iterator{this, len}; }
  };

int main() {
  Vektor<int> v{4};
  cout << "v: ";
  unsigned int count = 0;
  for (Vektor<int>::iterator it = v.begin(); it != v.end(); ++it) {
    *it = count++; cout << *it << " ";
  }
  cout << endl;
}
```

```
v: 0 1 2 3
```

Template Spezialisierung

⟨Templ'Spec'⟩ ⟶ “`template`” “<” [⟨Templ'Param's⟩] “>” ⟨Dekl'⟩

Templates können spezialisiert werden für konkrete Argumente. ⟨Dekl'⟩ ist hierbei eine Definition für eine Instanz eines bereits bekannten templates. Der Zweck ist i.d.R. effizientere Implementierungen für gewisse Spezialfälle bereitstellen zu können. Wenn ⟨Templ'Spec'⟩ wiederum einen nichtleeren Satz von Parametern akzeptiert spricht man von *partieller* Spezialisierung. Nur templates von Klassen und Variablen können partiell spezialisiert werden.

Beispiel (Spezialisierte 2-Norm). Wir spezialisieren die Implementierung von `pNorm` für $p = 2$.

array_2norm.cpp

```cpp
#include <iostream>
#include <array>
#include <cstddef>
#include <cmath>

using namespace std;

template<class T = double, unsigned int p = 2, size_t dim>
  T pNorm(const array<T, dim>& v) {
    T x = 0;
    for (const T& y: v)
      x += pow(y, p);
    return pow(x, static_cast<T>(1) / p);
}

template<>
  double pNorm<double, 2, 3>(const array<double, 3>& v) {
    return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
}

int main() {
  array<double, 3> a1{1, 2, 3};
  cout << pNorm<double, 2, 3>(a1) << endl;

  array<double, 5> a2{5, 4, 3, 2, 1};
  cout << pNorm<double, 3>(a2) << endl;
}
```

```
3.74166
6.0822
```

Beispiel. Wir implementieren `pNorm` als Template von Funktionsobjekten um es partiell spezialisieren zu können.

partial_pnorm.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <limits>
#include <array>
#include <vector>
#include <cstddef>
#include <cmath>

using namespace std;

template<class Vec, unsigned int p = 2>
  class PNorm{
    private:
      using X = typename Vec::value_type;
    public:
      X operator()(const Vec& v) {
```

```
        X x = 0;
        for (const X& y: v)
          x += pow(y, p);
        return pow(x, static_cast<X>(1) / p);
      }
  };

template<class Vec>
  class PNorm<Vec, 2>{
    private:
      using X = typename Vec::value_type;
    public:
      X operator()(const Vec& v) {
        X x = 0;
        for (const X& y: v)
          x += y * y;
        return sqrt(x);
      }
  };

int main() {
  vector<double> c{5, 6};
  array<float, 2> a{3, 4};

  cout
    << setprecision(numeric_limits<double>::digits10 + 1)
    << PNorm<vector<double>>{}(c) << endl
    << PNorm<array<float, 2>, 3>{}(a) << endl;
}
```

```
7.810249675906654
4.497941493988037
```

Substitution Failure is Not An Error (SFINAE)

Wenn, im Falle eines Templates mit partiellen Spezialisierungen, das Einsetzen der verbleibenden Argumente in der Spezialiserung scheitert, ist dies *kein* Kompilierfehler. Es wird stattdessen stillschweigend auf die Verwendung dieser Spezialisierung verzichtet und i.A. die nächste, weniger gut passende, Spezialisierung in Betracht gezogen. Dieses Verhalten wird gelegentlich explizit ausgenutzt um partielle Spezialisierungen einzuschränken auf nur bestimmte Argumente.

Die Standardlibrary stellt hierfür in der Header-Datei `type_traits` einige *Typprädikate* und Werkzeuge bereit.

Instanzen des Templates `std::enable_if<`B`, `T`>` sind Klassen, die einen Typalias `type` als Klassenkomponente mit Wert T enthalten, g.d.w. B den Wert `true` hat. `enable_if` ließe sich z.B. implementieren, wie folgt:

Implementierung enable_if

```
template<bool B, class T = void> class enable_if {};

template<class T> class enable_if<true, T> {
  public:
    using type = T;
};

template<bool B, class T = void>
  using enable_if_t = typename enable_if<B, T>::type;
```

Instanzen von Typprädikaten wie z.B. `is_integral<T>` sind Klassen, die einen statische Datenkomponente `value` vom Typ `bool` (stets mit Wert `true`) enthalten, g.d.W. der Typ T bestimmte, vom Prädikat abhängige, Einschränkungen erfüllt. Eine stark vereinfachte Implementierung von `is_integral` in der, statt eine Einschränkung zu prüfen, jeder das Prädikat erfüllende Typ eine eigene Instanz benötigen würde, könnte aussehen, wie folgt:

Implementierung is_integral

```
template<class T> class is_integral {};

template<> class is_integral<int> {
  public:
    static constexpr bool value = true;
};
```

Beispiel. Wir verwenden SFINAE um Spezialisierungen des Konstruktors einer Klasse `Approx` einzuschränken sodass die eine Spezialisierung nur für Parameter ganzzahligen Typs und die andere nur für Parameter von Gleitpunktzahlentyp gewählt werden.

sfinae_demo.cpp

```
#include <type_traits>
#include <iostream>
#include <cmath>

using namespace std;

class Approx {
  public:
    int x;

    template<class Integral, enable_if_t<is_integral<Integral>::value, bool> = true>
      Approx(Integral i): x(i) {}

    template<class Floating, enable_if_t<is_floating_point<Floating>::value, bool> = true>
      Approx(Floating d): x(round(d)) {}
};

int main() {
  Approx a1{3}, a2{5.7};

  cout << a1.x << " " << a2.x << endl;
```

```
}
```

```
3 6
```

Präprozessor-Direktiven

⟨Pre'Dir⟩ → “#define” ⟨ident⟩ [“(” ⟨ident⟩ {“,” ⟨ident⟩} “)”] ⟨text⟩
| “#ifdef” ⟨ident⟩ | “#ifndef” ⟨ident⟩ | “#else” | “#endif”
| “#include” (“"” ⟨file⟩ “"” | “<” ⟨ident⟩ “>”)

Bevor eine Übersetzungseinheit (i.d.R. eine `.cpp`-Datei) kompiliert wird, wird der Präprozessor auf dem Text der Übersetzungseinheit ausgeführt. Der Präprozessor nimmt textuelle Ersetzungen anhand von Direktiven der Form ⟨Pre'Dir⟩ vor.

`#define`

Die Direktive `#define` führt ein token ⟨ident⟩ ein. Im Folgenden wird dann jedes Vorkommen des eingeführten tokens durch ⟨text⟩ ersetzt. Das token zählt im Folgenden als *definiert*, auch wenn ⟨text⟩ leer ist.

Das eingeführte token kann parametrisiert werden. Es werden dann bei Ersetzung eines Vorkommen des eingeführten tokens mit den gegebenen Argumenten in ⟨text⟩ die Argumente eingesetzt wo immer die ⟨ident⟩ der Parameter darin vorkommen.

Es handelt sich hierbei um eine Ersetzung auf rein textuellem Niveau. Insb. Bindungsstärken von Operatoren werden, mit manchmal unintuitiven Folgen, nicht berücksichtigt.

Beispiel. Wir definieren die tokens `SEVENTEEN` und `PLUS`. `PLUS` ist hierbei parametrisiert über zwei Parameter `a` und `b`.

define_demo.cpp

```cpp
#include <iostream>
using namespace std;

#define SEVENTEEN (14 + 3)
#define PLUS(a, b) (a + b)

int main() {
  cout << SEVENTEEN << " "
       << PLUS(3, 14)
       << endl;
}
```

```
17 17
```

`#ifdef`, `#ifndef`, etc.

Die Direktive `#ifdef` bzw. `#ifndef` fügt den Text zwischen der Zeile die die Direktive enthält und dem darauf folgenden `#endif` bzw. `#else` ein, g.d.w. das angegebene ⟨ident⟩ definiert ist, bzw. nicht definiert ist. Auf ein etwaiges `#else` dürfen bis `#endif` weitere Zeilen Text folgen. Diese werden stattdessen eingefügt, falls die Bedingung bzgl. Definiertheit nicht zutrifft.

Beispiel. Wir definieren das token `SEVENTEEN` und fügen in das Hauptprogramm eine Zeile Code ein wenn `SEVENTEEN` definiert ist und stattdessen eine andere Zeile, wenn nicht.

ifdef_demo.cpp

```cpp
#include <iostream>
using namespace std;

#define SEVENTEEN 17

int main() {
#ifdef SEVENTEEN
  cout << SEVENTEEN << endl;
#else
  cout << 18 << endl;
#endif
}
```

```
17
```

`#include`

Die Direktive `#include` findet eine Datei im lokalen System anhand von ⟨file⟩ bzw. ⟨ident⟩ und ersetzt die Direktive durch den Inhalt jener Datei. Die Methode mit der der Compiler die Datei findet ist im Standard nicht festgelegt. I.d.R. sucht jedoch die Variante mit " im gleichen Verzeichnes wie die zu übersetzende `.cpp`-Datei und die Variante mit <, > i.d.R. in systemweiten Standardverzeichnissen.

Beispiel. Wir binden die Datei `iostream` aus einem nicht näher spezifizierten systemweiten Standardverzeichnis ein.

hello.cpp

```cpp
#include <iostream>

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
}
```

```
Hello, World!
```

Header-Dateien

Oft werden die *Deklarationen* von „Objekten“ (d.h. Funktionen, Klassen, globalen Variablen, etc.) ausgelagert in spezielle *Header-Dateien*, die i.d.R. nur Deklarationen enthalten und die Dateiendung `.h` tragen. Diese Header-Dateien werden dann unter Verwendung der Präprozessor-Direktive `#include` in i.A. mehrere Übersetzungseinheiten (i.d.R. `.cpp`-Dateien) eingebunden.

Double inclusion

Es kann leicht passieren, dass eine Header-Datei mehrfach in eine Übersetzungseinheit eingebunden wird. Es ist jedoch i.A. nicht zulässig beliebige „Objekte“ (insb. Klassen) mehrfach zu deklarieren bzw. zu definieren.

Beispiel. Die Übersetzungseinheit `child.cpp` bindet `grandparent.h` mittelbar doppelt ein. Die Übersetzung schlägt daher fehl.

`grandparent.h`

```
class Klasse {
  public: int attribut;
};
```

`parent.h`

```
#include "grandparent.h"
```

`child.cpp`

```
#include "grandparent.h"
#include "parent.h"

int main() {}
```

```
In file included from parent.h:1,
                 from child.cpp:2:
grandparent.h:1:7: error: redefinition of ‘class Klasse’
    1 | class Klasse {
      |       ^~~~~~
In file included from child.cpp:1:
grandparent.h:1:7: note: previous definition of ‘class Klasse’
    1 | class Klasse {
      |       ^~~~~~
```

Include guards

Zur Vermeidung des Problems der double inclusion hat sich historisch die Verwendung von *include guards* durchgesetzt. Es wird hierfür für jede Header-Datei ein token definiert und der gesamte Text der Header-Datei nur dann vom Präprozessor eingesetzt, wenn das jeweilige token noch nicht definiert ist. Nachteil hierbei ist

dass es dem Programmierer überlassen bleibt ein hinreichend eindeutiges token für jede Header-Datei zu wählen.

Beispiel. Wir vermeiden das Problem der double inclusion durch Verwendung von include guards.

grandparent_guard.h

```
#ifndef GRANDPARENT_GUARD_H
#define GRANDPARENT_GUARD_H

class Klasse {
  public: int attribut;
};
#endif
```

parent_guard.h

```
#include "grandparent_guard.h"
```

child_guard.cpp

```
#include "grandparent_guard.h"
#include "parent_guard.h"

int main() {}
```

`#pragma once`

Bei den `#pragma` Präprozessor-Direktiven handelt es sich um Compiler-spezifische Erweiterungen des Präprozessors. Der Standard definiert die Wirkung von `#pragma`-Direktiven nicht.

I.W. alle Compiler unterstützen `#pragma once`. Die Wirkung ist analog zu include guards; Header-Dateien, die die Direktive enthalten, werden in jeder Übersetzungseinheit maximal einmal eingebunden.

Der Vorteil gegenüber include guards liegt in der Beseitigung einiger Fehlerquellen wie der sonst benötigten `#endif` Direktive und der Wahl eines hinreichend eindeutigen tokens.

Beispiel. Wir vermeiden das Problem der double inclusion durch Verwendung von `#pragma once`.

grandparent_pragma.h

```
#pragma once

class Klasse {
  public: int attribut;
};
```

parent_pragma.h

```
#include "grandparent_pragma.h"
```

child_pragma.cpp

```
#include "grandparent_pragma.h"
#include "parent_pragma.h"

int main() {}
```

Separate Compilation

Objekt-Dateien

Das Kompilieren einer Übersetzungseinheit gliedert sich in mehrere Phasen. Zunächst wendet der Präprozessor textuelle Ersetzungen an, dann wird der resultierende Text als C++-Code kompiliert zu einer *Objekt-Datei* und dann die resultierende Objekt-Datei zu einer ausführbaren Binärdatei weiterverarbeitet. Durch Übergeben des Kommandozeilenparameters `-c` an `g++` (z.B. also `g++ -c hello.cpp`) kann das Weiterverarbeiten in eine ausführbare Binärdatei unterdrückt werden. Es wird dann eine Objekt-Datei mit dem gleichen Namen wie die übersetzte `.cpp`-Datei, jedoch mit Dateiendung `.o` produziert (z.B. also `hello.o`).

Die produzierte Objekt-Datei enthält kompilierten Maschinencode und einige Metadaten bzgl. insb. Funktionen und globalen Variablen. Es korrespondiert ein *Symbol* der Objekt-Datei zu jedem „Objekt“, d.h. insb. jeder Funktion und globalen Variablen.

Linker

Der *Linker* wird, bei Aufruf ohne den Kommandozeilenparameter `-c`, vom Befehl `g++` automatisch ausgeführt. Er verknüpft Erwähnungen von Symbolen in Objekt-Dateien mit den Implementierungen selbiger, i.A. auch über mehrere Objekt-Dateien, und produziert eine ausführbare Binärdatei. Werden Symbole erwähnt, die sich in keiner der angegebenen Objekt-Dateien finden lässt, so scheitert die Ausführung des Linkers mit einer Fehlermeldung.

Beispiel. Wir kompilieren eine leere Datei. Die Fehlermeldung die wir dabei vom Linker `ld` erhalten besagt, dass das Symbol `main` zwar erwähnt wurde (es wird zum Erstellen einer ausführbaren Binärdatei natürlich zwingend benötigt), jedoch nicht aufzufinden ist.

```
…/bin/ld: …/lib/crt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Auf eine Liste von Objekt-Dateien $l_1, \ldots, l_n$ lässt sich der Linker aufrufen als `g++` $l_1 \ \ldots \ l_n$. Die Reihenfolge in der die Objekt-Dateien als Kommandozeilenparameter angegeben werden ist i.A. relevant. Ein oft zielführen-

der Ansatz ist es, dass Objekt-Dateien, die ein Symbol *s* verwenden, im Befehl vorkommen müssen vor der Objekt-Datei die *s* definiert.

Beim Linken von C++-Code der Symbole verwendet, die in anderen Übersetzungseinheiten implementiert sind, ist eine Implementierung des Symbols in Form einer Objekt-Datei zwar notwendig, aber i.A. nicht hinreichend. Es wird i.d.R. auch die Deklaration des „Objekts“ benötigt. Hierfür wird die Deklaration i.d.R. in einer Header-Datei ausgelagert und vermöge `#include` in den C++-Code eingebunden, der das Symbol verwendet.

Von der Verwendung von Anweisungen wie `using namespace` in Header-Dateien ist abzuraten, da hierduch alle Symbole aus dem verwendeten `namespace` auch in Übersetzungseinheiten sichtbar würden, die die Header-Datei bloß mit `#include` einbinden. Man spricht bei Verletzung dieses Prinzips von *namespace pollution*.

Beispiel (Programmbibliothek). Wir demonstrieren die konventionelle Struktur einer Programmbibliothek indem wir eine Funktion `greet` zur Ausgabe eines `string`s auf der Standardausgabe separat implementieren von ihrer Verwendung in den Übersetzungseinheiten `hello1.cpp` und `hello2.cpp`.

`greet.h`

```
#include <string>

void greet(std::string greeting);
```

`greet.cpp`

```
#include <iostream>
#include <string>
#include "greet.h"

using namespace std;

void greet(string greeting) {
  cout << greeting << endl;
}
```

`hello1.cpp`

```
#include <string>

using namespace std;

void greet(string greeting);

int main() {
  greet("Hello, World!");
}
```

`hello2.cpp`

```
#include "greet.h"

int main() {
```

```
  greet("Hello!");
}
```

Die Übersetzung von `hello1.cpp` (und analog für `hello2.cpp`) kann erfolgen, wie folgt:

```
g++ -c greet.cpp
g++ -c hello1.cpp
g++ hello1.o greet.o
./a.out
```

Die durch die Übersetzung von `hello1.cpp` erzeugte ausführbare Binärdatei erzeugt bei Ausführung die folgende Ausgabe:

```
Hello, World!
```

`hello2.cpp` produziert bei Ausführung:

```
Hello!
```