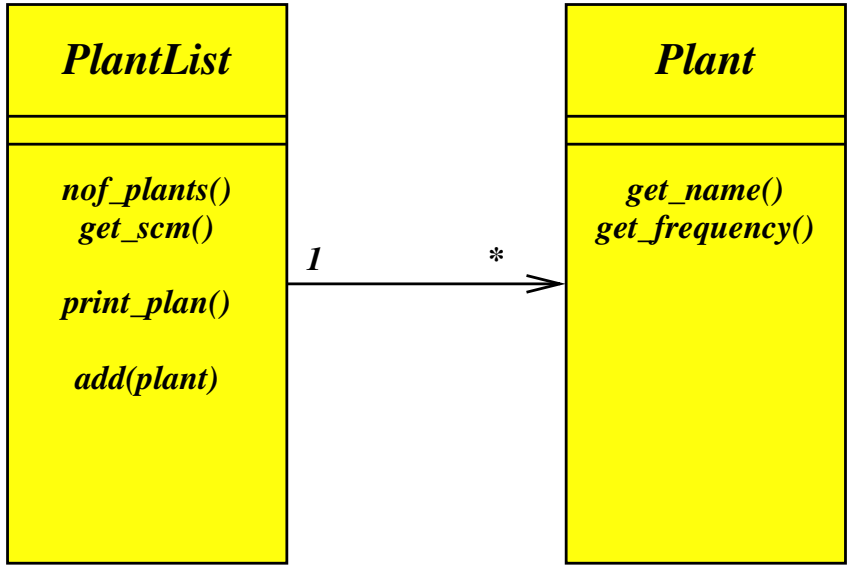

- Die Aufgabenstellung ist die Generierung eines tageweisen Bewässerungsplans für eine Menge von Pflanzen, von denen jede unterschiedliche Bewässerungsfrequenzen bevorzugt.
- *scm* steht für das *kleinste gemeinsame Vielfache* (engl. *smallest common multiple*; kurz: kgV) und *get_scm* liefert das kgV aller erfasster Bewässerungsfrequenzen zurück. Nach dieser Zahl von Tagen wiederholt sich der Bewässerungsplan.

Plant.hpp

```
#ifndef PLANT_H
#define PLANT_H

#include <string>

class Plant {
   public:
      // constructors
      Plant(const std::string& plantName, int wateringFrequency);
         // PRE: wateringFrequency >= 1
      Plant(const Plant& plant);

      // accessors
      const std::string& get_name() const;
      int get_frequency() const;

   private:
      const std::string name;
      int frequency;
};

#endif
```

Plant.cpp

```
#include <cassert>
#include "Plant.hpp"

using std::string;

Plant::Plant(const string& plantName, int wateringFrequency) :
      name{plantName}, frequency{wateringFrequency} {
   assert(wateringFrequency >= 1);
}

Plant::Plant(const Plant& plant) :
      name{plant.name}, frequency{plant.frequency} {
}

const string& Plant::get_name() const {
   return name;
}

int Plant::get_frequency() const {
   return frequency;
}
```

Plant.hpp

```
class Plant {
   public:
      // constructors
      Plant(const std::string& plantName, int wateringFrequency);
         // PRE: wateringFrequency >= 1
      // ...

   private:
      const std::string name;
      int frequency;
};
```

Plant.cpp

```
Plant::Plant(const string& plantName, int wateringFrequency) :
      name{plantName}, frequency{wateringFrequency} {
   assert(wateringFrequency >= 1);
}
```

- Wenn sich Variablen wie hier *name* nie verändern, dann ist es sinnvoll, sie mit **const** zu vereinbaren. Solche Variablen müssen dann zwingend über einen ⟨ctor-initializer⟩ initialisiert werden.

PlantList.hpp

```
#ifndef PLANTLIST_H
#define PLANTLIST_H

#include <list>
#include "Plant.hpp"

class PlantList {
   public:
      PlantList();

      int nof_plants() const;
      int get_scm() const; // PRE: nof_plants() > 0

      void print_plan(int day); // PRE: day >= 0
      void print_plan();

      void add(Plant plant);

   private:
      std::list<Plant> plants;
      int scm; // of watering frequencies
};

#endif
```

PlantList.hpp

```
#include <list>
#include "Plant.hpp"

// ...

std::list<Plant> plants;
```

- Dies deklariert *plants* als eine Liste von Elementen des Typs *Plant*.
- *std*::*list* ist eine Template-Klasse, bei der der Elementtyp als Typparameter übergeben wird.
- Entsprechend wird hier nicht nur *plants* deklariert, sondern auch implizit eine neue Klasse ausgehend von *std*::*list* erzeugt. Dies wird auch als Instantiierung eines Templates bezeichnet (*template instantiation*).
- Diese Template-Klasse gehört zur STL (*standard template library*), die Bestandteil von ISO C++ ist.

PlantList.cpp

```
void PlantList::print_plan(int day) {
   assert(day >= 0);
   for (Plant& plant: plants) {
      if (day % plant.get_frequency() == 0) {
         std::cout << plant.get_name() << std::endl;
      }
   }
}
```

- Seit C++11 gibt es eine spezielle Form der **for**-Schleife, die die Iteration einer Datenstruktur auf elegantem Wege erlaubt.
- In jedem Schleifendurchlauf ist *plant* eine Referenz auf das aktuelle Element aus der Liste *plants*.

```
for (list<Plant>::iterator iterator = plants.begin();
      iterator != plants.end(); ++iterator) {
   if (day % iterator->get_frequency() == 0) {
      cout << iterator->get_name() << endl;
   }
}
```

- Die elegante Form der **for**-Schleife wird intern durch Iteratoren umgesetzt, die ähnlich wie Array-Zeiger verwendet werden können, da all die entsprechenden Operatoren für sie überladen sind.
- *list*<*Plant*>::*iterator* ist eine Klasse, die innerhalb der Klasse *list*<*Plant*> enthalten ist.
- Innerhalb des Initialisierungsteils der **for**-Schleife wird hier *iterator* als Objekt dieser speziellen Iterator-Klasse deklariert und mit *plants.begin*() initialisiert, das einen auf das erste Element zeigenden Iterator zurückliefert.

PlantList.cpp

```
void PlantList::add(Plant plant) {
   int frequency = plant.get_frequency();
   if (scm == 0) {
      scm = frequency;
   } else if (scm % frequency != 0) {
      // computing smallest common multiple using Euclid
      int x0 = scm, x = scm, y0 = frequency, y = frequency;
      while (x != y) {
         if (x > y) {
            y += y0;
         } else {
            x += x0;
         }
      }
      scm = x;
   }
   plants.push_back(plant);
}
```

- *plants.push_back*(*plant*) belegt Speicher für eine Kopie von *plant* und hängt diese Kopie an das Ende der Liste ein.

Hauptprogramm

WateringPlan.cpp

```
#include <iostream>
#include <string>
#include "Plant.hpp"
#include "PlantList.hpp"

int main() {
   PlantList plants;
   std::string name; int frequency;

   while (std::cin >> name && std::cin >> frequency) {
      plants.add(Plant{name, frequency});
   }
   plants.print_plan();
}
```

WateringPlan.cpp

```
while (std::cin >> name && std::cin >> frequency) {
   plants.add(Plant{name, frequency});
}
```

- Normalerweise liefert *cin* >> *name* den Wert von *cin* zurück, um eine Verkettung von Eingabe-Operationen für den gleichen Eingabestrom zu ermöglichen.
- Hier jedoch findet implizit eine Konvertierung statt, da ein **bool**-Wert benötigt wird. Dies gelingt u.a. mit Hilfe eines sogenannten Konvertierungs-Operators der entsprechenden Klasse.
- Entsprechend ist die gesamte Bedingung genau dann wahr, falls beide Lese-Operationen erfolgreich sind.
- *Plant*{*name*, *frequency*} erzeugt ein sogenanntes temporäres Objekt des Typs *Plant*, das vollautomatisch wieder aufgeräumt wird, sobald die Ausführung der zugehörigen Anweisung beendet ist.

- In allen bisherigen Beispielen belegten die Objekte entweder statischen Speicherplatz oder sie lebten auf dem Stack.
- Dies vermied bislang völlig den Aufwand einer dynamischen Speicherverwaltung. Es gehört zu den Vorteilen von C++ (und einigen anderen hybriden OO-Sprachen), dass nicht für alle Objekte der Speicherplatz dynamisch belegt werden muss.

- Auf der anderen Seite ist die Beachtung einiger Richtlinien unerlässlich, wenn Klassen Zeiger auf dynamische Datenstrukturen verwenden, da
  - wegen der fehlenden automatischen Speicherbereinigung (*garbage collection*) es in der Verantwortung der Klassenimplementierung liegt, referenzierte Datenstrukturen wieder freizugeben und da
  - Konstruktoren und Zuweisungs-Operatoren per Voreinstellung nur die Datenfelder kopieren (*shallow copy*) und somit Zeigerwerte implizit vervielfältigt werden können.

ListOfFriends.hpp

```
class ListOfFriends {
   public:
      // constructor
      ListOfFriends();
      ListOfFriends(const ListOfFriends& other);
      ListOfFriends(ListOfFriends&& other);
      ~ListOfFriends();

      // assignment
      ListOfFriends& operator=(const ListOfFriends& list);
      ListOfFriends& operator=(ListOfFriends&& list);

      // printing
      void print();

      // mutator
      void add(const Friend& f);

   private:
      struct Node* root;
      void addto(Node*& p, Node* newNode);
      void visit(const Node* const p);
}; // class ListOfFriends
```

- Ein Objekt der Klasse *ListOfFriends* verwaltet eine Liste von Freunden und ermöglicht die sortierte Ausgabe (alphabetisch nach dem Namen).
- Die Implementierung beruht auf einem sortierten binären Baum. Der Datentyp **struct** *Node* repräsentiert einen Knoten dieses Baums.
- Zu beachten ist hier, dass eine Deklaration eines Objekts des Typs **struct** *Node*∗ auch dann zulässig ist, wenn **struct** *Node* noch nicht bekannt ist, da der benötigte Speicherplatz bei Zeigern unabhängig vom referenzierten Datentyp ist.

ListOfFriends.cpp

```
struct Node {
   struct Node* left;
   struct Node* right;
   Friend f;
   Node(const Friend& newFriend);
   Node(const Node* const& node);
   ~Node();
}; // struct Node

Node::Node(const Friend& newFriend) :
      left{nullptr}, right{nullptr}, f{newFriend} {
} // Node::Node
```

- Im Vergleich zu **class** sind bei **struct** alle Komponenten implizit **public**. Da hier die Datenstruktur nur innerhalb der Implementierung deklariert wird, stört dies nicht, da sie von außen nicht einsehbar ist.
- Der hier gezeigte Konstruktor legt ein Blatt an.

ListOfFriends.cpp

```
Node::Node(const Node* const& node) :
      left{nullptr}, right{nullptr}, f{node->f} {
   if (node->left) {
      left = new Node{node->left};
   }
   if (node->right) {
      right = new Node{node->right};
   }
} // Node::Node
```

- Der zweite Konstruktor für **struct** *Node* akzeptiert einen Zeiger auf *Node* als Parameter. Die beiden **const** in der Signatur stellen sicher, dass nicht nur der (als Referenz übergebene) Zeiger nicht verändert werden darf, sondern auch nicht der Knoten, auf den dieser verweist.
- Hier ist es sinnvoll, einen Zeiger als Parameter zu übergeben, da in diesem Beispiel Knoten ausschließlich über Zeiger referenziert werden.

⟨new-expression⟩	⟶	[„::“] **new** [⟨new-placement⟩]
		⟨new-type-id⟩ [⟨new-initializer⟩]
	⟶	[„::“] **new** [⟨new-placement⟩]
		„(“ ⟨type-id⟩ „)“ [⟨new-initializer⟩]
⟨new-placement⟩	⟶	„(“ ⟨expression-list⟩ „)“
⟨new-type-id⟩	⟶	⟨type-specifier-seq⟩
	⟶	⟨new-declarator⟩
⟨new-declarator⟩	⟶	⟨ptr-operator⟩ [⟨new-declarator⟩]
	⟶	⟨noptr-new-declarator⟩
⟨noptr-new-declarator⟩	⟶	„[“ ⟨expression⟩ „]“
		[⟨attribute-specifier-seq⟩]
	⟶	⟨noptr-new-declarator⟩
		„[“ ⟨constant-expression⟩ „]“
		[⟨attribute-specifier-seq⟩]
⟨new-initializer⟩	⟶	„(“ [⟨expression-list⟩] „)“
	⟶	⟨braced-init-list⟩

- Hier werden die Felder *left* und *right* zunächst in der Initialisierungssequenz auf **nullptr** initialisiert und nachher bei Bedarf auf neu angelegte Knoten umgebogen. So ist garantiert, dass die Zeiger immer wohldefiniert sind.
- Tests wie **if** (*node*−>*left*) überprüfen, ob ein Zeiger ungleich **nullptr** ist.
- Zu beachten ist hier, dass der Konstruktor sich selbst rekursiv für die Unterbäume *left* und *right* von *node* aufruft, sofern diese nicht **nullptr** sind.
- Auf diese Weise erhalten wir hier eine tiefe Kopie (*deep copy*), die den gesamten Baum beginnend bei *node* dupliziert.

ListOfFriends.cpp

```
Node::~Node() {
   delete left; delete right;
} // Node::~Node
```

- Wie beim Konstruieren muss hier die Destruktion bei *Node* rekursiv arbeiten.
- **delete** unternimmt nichts, wenn der angegebene Zeiger den Wert **nullptr** hat. Auf diese Weise wird die Rekursion beendet.
- Diese Lösung geht davon aus, dass ein Unterbaum niemals mehrfach referenziert wird.
- Nur durch die Einschränkung der Sichtbarkeit kann dies auch garantiert werden.

$$\langle\text{delete-expression}\rangle \longrightarrow [\ \text{„::“}\ ]\ \textbf{delete}\ \langle\text{cast-expression}\rangle$$
$$\longrightarrow [\ \text{„::“}\ ]\ \textbf{delete}\ \text{„[“}\ \text{„]“}\ \langle\text{cast-expression}\rangle$$

ListOfFriends.cpp

```
ListOfFriends::ListOfFriends() :
      root{nullptr} {
} // ListOfFriends::ListOfFriends

ListOfFriends::ListOfFriends(const ListOfFriends& other) :
      root{nullptr} {
   Node* r(other.root);
   if (r) {
      root = new Node (r);
   }
} // ListOfFriends::ListOfFriends
```

- Der Konstruktor ohne Parameter (*default constructor*) ist trivial: Wir setzen nur *root* auf **nullptr**.
- Der kopierende Konstruktor ist ebenso hier recht einfach, da die entscheidende Arbeit an den rekursiven Konstruktor für *Node* delegiert wird.
- Es ist hier nur darauf zu achten, dass der Konstruktor für *Node* nicht in dem Falle aufgerufen wird, wenn *list.root* gleich **nullptr** ist.

ListOfFriends.cpp

```
ListOfFriends::ListOfFriends(ListOfFriends&& other) : root{other.root} {
   other.root = nullptr;
} // ListOfFriends::ListOfFriends
```

- Der Übernahmekonstruktor (*move constructor*) wird ähnlich wie beim kopierenden Konstruktor implizit aufgerufen.
- Im Unterschied zum kopierenden Konstruktor entfällt jedoch die Notwendigkeit, die dynamische Datenstruktur zu duplizieren. Stattdessen darf sie übernommen werden, weil das Quellobjekt unmittelbar danach abgebaut wird.
- Es darf dabei nicht vergessen werden, dass Quellobjekt in einem Zustand zu hinterlassen, das einen Abbau zulässt. Entsprechend muss hier *other.root* auf **nullptr** gesetzt werden, um eine Freigabe der Datenstruktur zu verhindern.

ListOfFriends.cpp

```
ListOfFriends::~ListOfFriends() {
   delete root;
} // ListOfFriends::~ListOfFriends
```

- Analog delegiert der Destruktor für *ListOfFriends* die Arbeit an den Destruktor für *Node*.

ListOfFriends.cpp

```
ListOfFriends& ListOfFriends::operator=(const ListOfFriends& list) {
   if (this != &list) { // protect against self-assignment
      delete root;
      if (list.root) {
         root = new Node{list.root};
      } else {
         root = nullptr;
      }
   }
   return *this;
} // ListOfFriends::operator=
```

- Ein rekursiv arbeitender kopierender Konstruktur und zugehöriger Destruktor genügen alleine nicht, da der voreingestellte Zuweisungs-Operator nur den Wurzelzeiger kopieren würde (*shallow copy*) und eine rekursive Kopie (*deep copy*) unterbleiben würde.

- Dies würde die wichtige Annahme (des Destruktors) verletzen, dass der selbe Baum nicht von mehreren Objekten des Typs *ListOfFriends* referenziert werden darf.
- Entsprechend ist die Implementierung unvollständig, solange eine simple Zuweisung von *ListOfFriends*-Objekten diese wichtige Annahme verletzen kann.
- Bei der Implementierung des Zuweisungs-Operators ist darauf zu achten, dass Objekte an sich selbst zugewiesen werden können. **this** repräsentiert einen Zeiger auf das Objekt, auf der die aufgerufene Methode arbeitet. &*list* ermittelt die Adresse von *list* und erlaubt somit einen Vergleich von Zeigerwerten.

ListOfFriends.cpp

```
ListOfFriends& ListOfFriends::operator=(ListOfFriends&& list) {
   if (this != &list) { // protect against self-assignment
      delete root;
      root = list.root;
      list.root = nullptr;
   }
   return *this;
}
```

- Analog zum übernehmenden Konstruktor kann auch ein entsprechender Zuweisungsoperator angeboten werden, der implizit verwendet wird, wenn die rechte Seite anschließend abgebaut wird.

ListOfFriends.cpp

```
void ListOfFriends::addto(Node*& p, Node* newNode) {
   if (p) {
      if (newNode->f.get_name() < p->f.get_name()) {
         addto(p->left, newNode);
      } else {
         addto(p->right, newNode);
      }
   } else {
      p = newNode;
   }
} // ListOfFriends::addto

void ListOfFriends::add(const Friend& f) {
   Node* node = new Node(f);
   addto(root, node);
} // ListOfFriends::add
```

- Wenn ein neuer Freund in die Liste aufgenommen wird, ist ein neues Blatt anzulegen, das auf rekursive Weise in den Baum mit Hilfe der privaten Methode *addto* eingefügt wird.

ListOfFriends.cpp

```
void ListOfFriends::visit(const Node* const p) {
   if (p) {
      visit(p->left);
      cout << p->f.get_name() << ": " << p->f.get_info() << endl;
      visit(p->right);
   }
} // ListOfFriends::visit

void ListOfFriends::print() {
   visit(root);
} // ListOfFriends::print
```

- Analog erfolgt die Ausgabe rekursiv mit Hilfe der privaten Methode *visit*.

TestFriends.cpp

```
ListOfFriends list1;
```

- Diese Deklaration ruft implizit den Konstruktor von *ListOfFriends* auf, der keine Parameter verlangt (*default constructor*). In diesem Falle wird *root* einfach auf **nullptr** gesetzt werden.

TestFriends.cpp

```
ListOfFriends list2{list1};
```

- Diese Deklaration führt zum Aufruf des kopierenden Konstruktors, der den vollständigen Baum von *list1* für *list2* dupliziert.

TestFriends.cpp

```
ListOfFriends list3;
list3 = list1;
```

- Hier wird zunächst der Konstruktor von *ListOfFriends* ohne Parameter aufgerufen (*default constructor*).
- Danach kommt es zur Ausführung des Zuweisungs-Operators, der den Baum von *list1* dupliziert und bei *list3* einhängt.

TestFriends.cpp

```
ListOfFriends gen_friends() {
   ListOfFriends list;
   list.add(Friend{"Ralf", "lives in Neu-Ulm"});
   list.add(Friend{"Lisa", "loves her bike"});
   return list;
}

int main() {
   // ...
   ListOfFriends list4;
   list4 = gen_friends();
   // ...
}
```

- Hier wird zunächst der Konstruktor von *ListOfFriends* ohne Parameter aufgerufen (*default constructor*).
- Danach kommt es zur Ausführung des übernehmenden Zuweisungs-Operators, der die Datenstruktur der zurückgegebenen lokalen Variablen übernimmt, ohne diese zu duplizieren.