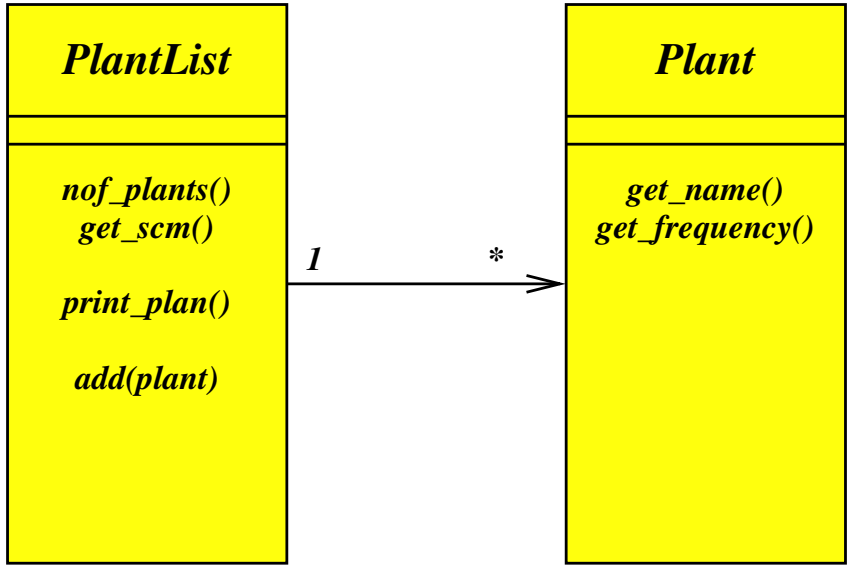

- Die Aufgabenstellung ist die Generierung eines tageweisen Bewässerungsplans für eine Menge von Pflanzen, von denen jede unterschiedliche Bewässerungsfrequenzen bevorzugt.
- *scm* steht für das *kleinste gemeinsame Vielfache* (engl. *smallest common multiple*; kurz: kgV) und *get_scm* liefert das kgV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

Plant.h

```
#ifndef PLANT_H
#define PLANT_H

#include <string>
using namespace std;

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

      // accessors
      string get_name() const;
      int get_frequency() const;

   private:
      string name;
      int frequency;
};

#endif
```

Plant.C

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

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

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

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

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

Plant.C

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

- Dies ist ein kopierender Konstruktor (*copy constructor*), der eine Initialisierung mit Hilfe eines bereits existierenden Objekts durchführt.
- Dieser Konstruktor wird in vielen Fällen implizit aufgerufen. Dazu ist beispielsweise der Fall, wenn dieser Datentyp im Rahmen der (noch vorzustellenden) Template-Klasse für Listen verwendet wird, da diese die Werte grundsätzlich kopiert.

Plant.C

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

- Deswegen ist dieser Konstruktor nicht nur eine Bereicherung der *Plant*-Klasse, sondern auch generell eine Voraussetzung, um Objekte dieser Klasse in Listen aufnehmen zu können.
- Wenn kein kopierender Konstruktor zur Verfügung gestellt wird (und auch sonst keine anderen Konstruktoren explizit deklariert werden) gibt es eine Voreinstellung für den kopierenden Konstruktor, der alle Felder einzeln kopiert. Dies wäre hier kein Problem, aber das kann zur bösen Falle werden, wenn Zeiger auf diese Weise vervielfältigt und dann später möglicherweise mehrfach freigegeben werden.

Plant.C

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

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

- Zu beachten ist hier das Schlüsselwort **const** am Ende der Signatur. Dies legt fest, dass diese Methode den (abstrakten!) Status des Objekts nicht verändert.

PlantList.h

```
#ifndef PLANTLIST_H
#define PLANTLIST_H

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

class PlantList {
   public:
      // constructors
      PlantList();
      // accessors
      int nof_plants() const;
      int get_scm() const; // PRE: nof_plants() > 0
      // printing
      void print_plan(int day); // PRE: day >= 0
      void print_plan();
      // mutators
      void add(Plant plant);
   private:
      list< Plant > plants;
      int scm; // of watering frequencies
};
#endif
```

PlantList.h

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

// ...

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

- Dies deklariert *plants* als eine Liste von Elementen des Typs *Plant*.
- *list* ist ein Template, das einen Typparameter als Elementtyp erwartet.
- Entsprechend wird hier nicht nur *plants* deklariert, sondern auch implizit ausgehend von dem Template *list* eine neue Klasse erzeugt. Dies wird auch als Instantiierung eines Templates bezeichnet (*template instantiation*).
- Das Listen-Template gehört zur STL (*standard template library*), die Bestandteil von ISO C++ ist.

PlantList.C

```
#include <cassert>
#include "PlantList.h"

using namespace std;

PlantList::PlantList() :
   scm(0) {
} // PlantList::PlantList

int PlantList::nof_plants() const {
   return plants.size();
} // PlantList::nof_plants

int PlantList::get_scm() const {
   assert(scm > 0);
   return scm;
} // PlantList::get_scm
```

- *scm* ist dank des Konstruktors immer wohldefiniert.
- Später bestehen wir (entsprechend der Vorbedingung) darauf, dass mindestens eine Pflanze eingetragen ist, bevor ein Aufruf von *get_scm* zulässig ist.

PlantList.C

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

- *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.
- Die **for**-Schleife endet, sobald *iterator* den Wert von *plants.end*() erreicht hat. In diesem Falle sind alle in der Liste enthaltenen Objekte bereits betrachtet worden.

PlantList.C

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

- *iterator* ist wie ein Zeiger zu verwenden. Aus diesem Grunde wird »−>« an Stelle von ».« für den Aufruf einer Methode auf dem gerade referenzierten Objekt zu verwenden.
- Mit ++*iterator* wandert der Iterator zum nächsten Element in der Liste.

PlantList.C

```
void PlantList::print_plan() {
   for (int day(0); day < scm; day += 1) {
      cout << "Day " << day << ":" << endl;
      print_plan(day);
   }
} // PlantList::print_plan
```

- Prinzipiell wäre es hier sinnvoll, auch die beiden Methoden namens *print_plan* mit *const* zu deklarieren. Dies ginge allerdings nur, wenn in der Methode, die den Iterator verwendet, *const_iterator* an Stelle von *iterator* verwenden würde.

PlantList.C

```
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);
} // PlantList::add
```

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

WateringPlan.C

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

using namespace std;

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

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

WateringPlan.C

```
while (cin >> name && 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 anonymes 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.

Integer.h

```
#ifndef INTEGER_H
#define INTEGER_H

class Integer {
   public:
      // constructor
      Integer(int initval);
      // destructor
      ~Integer();
      // accessor
      int get_value() const;
      void set_value(int newval);
   private:
      int value;
}; // class Integer

#endif
```

- Die Signatur eines Destruktors besteht aus einer Tilde »~«, dem Namen der Klasse (analog zu den Konstruktoren) und einer leeren Parameterliste.

Integer.C

```
#include <iostream>
#include "Integer.h"

using namespace std;

Integer::Integer(int intval) :
   value(intval) {
   cout << "Integer constructor: value = " <<
      value << endl;
} // Integer::Integer

Integer::~Integer() {
   cout << "Integer destructor: value = " <<
      value << endl;
} // Integer::~Integer

int Integer::get_value() const {
   return value;
} // Integer::get_value

void Integer::set_value(int newval) {
   value = newval;
} // Integer::set_value
```

- Prinzipiell können (wie in diesem Beispiel) beliebige Anweisungen wie auch Ausgaben in Destruktoren aufgenommen werden. Das kann aber zusätzliche Komplikationen mit sich bringen, wenn etwa eine Ausnahmenbehandlung in Gang gesetzt werden sollte.
- Wenn kein Destruktor angegeben wird, dann kommt eine Voreinstellung zum Zuge, die die Destruktoren für alle einzelnen Datenfelder aufruft.

Implizite Aufrufe von Konstruktoren und Destruktoren 149

TestInteger1.C

```
#include <iostream>
#include "Integer.h"

using namespace std;

int main() {
   cout << "main starts" << endl;
   {
      Integer i(1);
      cout << "working on i = " << i.get_value() << endl;
   }
   cout << "main ends" << endl;
} // main
```

- Durch eine Deklaration wie hier mit *Integer i*(1) wird der passende Konstruktor implizit aufgerufen.

Implizite Aufrufe von Konstruktoren und Destruktoren 150

```
dublin$ TestInteger1
main starts
Integer constructor: value = 1
working on i = 1
Integer destructor: value = 1
main ends
dublin$
```

- Der Sichtbereich von *i* ist statisch begrenzt auf den umgebenden Block. Die Lebenszeit beginnt und endet mit der Laufzeit des umgebenden Blocks.
- Der Destruktor wird implizit beim Verlassen des Blocks aufgerufen.

TestInteger2.C

```
#include <iostream>
#include "Integer.h"
using namespace std;

int main() {
   cout << "main starts" << endl;

   Integer* ip (new Integer(1));
   cout << "working on ip = " << ip->get_value() << endl;
   delete ip;

   cout << "main ends" << endl;
} // main
```

- Mit *Integer∗ ip* wird *ip* als Zeiger auf *Integer* deklariert.
- Der Ausdruck **new** *Integer*(1) veranlasst das dynamische Belegen von Speicher für ein Objekt des Typs *Integer* und ruft den passenden Konstruktor auf.

```
dublin$ TestInteger2
main starts
Integer constructor: value = 1
working on ip = 1
Integer destructor: value = 1
main ends
dublin$
```

- Die Verantwortung für die Speicherfreigabe verbleibt beim Aufrufer des **new**-Operators.
- Mit **delete** *ip* wird der Destruktor von *ip* aufgerufen und danach der belegte Speicherplatz freigegeben.

TestInteger3.C

```
#include <iostream>
#include "Integer.h"

using namespace std;

int main() {
   cout << "main starts" << endl;

   Integer* ip (new Integer(1));
   Integer& ir (*ip);
   cout << "working on ip = " << ip->get_value() << endl;
   ir.set_value(2);
   cout << "working on ip = " << ip->get_value() << endl;
   delete ip;

   cout << "main ends" << endl;
} // main
```

- Mit *Integer& ir* (∗*ip*) wird *ir* als Referenz für ein *Integer*-Objekt deklariert.

- Im Vergleich zu Zeigern gibt es bei Referenzen einige Unterschiede:
  - Im Rahmen ihrer Deklaration müssen sie mit einem Objekt des referenzierten Typs verbunden werden.
  - Referenzen bleiben konstant, d.h. sie können während ihrer Lebenszeit nicht ein anderes Objekt referenzieren.
  - Referenzen werden syntaktisch wie das Objekt, das sie referenzieren, behandelt. Entsprechend wird etwa ».« an Stelle von »−>« verwendet.

ListOfFriends.h

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

      // overloaded operators
      ListOfFriends &operator=(const 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.C

```
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(0), right(0), 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.C

```
Node::Node(const Node* const& node) :
   left(0), right(0), 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 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.

- Hier werden die Felder *left* und *right* zunächst in der Initialisierungssequenz auf 0 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 0 ist.
- Zu beachten ist hier, dass der Konstruktor sich selbst rekursiv für die Unterbäume *left* und *right* von *node* aufruft, sofern diese nicht 0 sind.
- Auf diese Weise erhalten wir hier eine tiefe Kopie (*deep copy*), die den gesamten Baum beginnend bei *node* dupliziert.

ListOfFriends.C

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

- Wie beim Konstruieren muss hier die Destruktion bei *Node* rekursiv arbeiten.
- Diese Lösung geht davon aus, dass ein Unterbaum niemals mehrfach referenziert wird.
- Nur durch die Einschränkung der Sichtbarkeit kann dies auch garantiert werden.

ListOfFriends.C

```
ListOfFriends::ListOfFriends() :
   root(0) {
} // ListOfFriends::ListOfFriends

ListOfFriends::ListOfFriends(const ListOfFriends& list) :
   root(0) {
   Node* r(list.root);
   if (r) {
      root = new Node (r);
   }
} // ListOfFriends::ListOfFriends
```

- Der Konstruktor ohne Parameter (*default constructor*) ist trivial: Wir setzen nur *root* auf 0.
- 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* ungleich 0 ist.

ListOfFriends.C

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

- Analog delegiert der Destruktor für *ListOfFriends* die Arbeit an den Destruktor für *Node*.
- Es ist nicht schlimm, wenn der **delete**-Operator für 0-Zeiger aufgerufen wird. Das wird vom ISO-Standard für C++ ausdrücklich erlaubt. Die **if**-Anweisung spart aber Ausführungszeit.

ListOfFriends.C

```
ListOfFriends& ListOfFriends::operator=
      (const ListOfFriends& list) {
   if (this != &list) { // protect against self-assignment
      if (root) {
         delete root;
      }
      if (list.root) {
         root = new Node (list.root);
      } else {
         root = 0;
      }
   }
   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.C

```
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.C

```
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.C

```
ListOfFriends list1;
```

- Diese Deklaration ruft implizit den Konstruktor von *ListOfFriends* auf, der keine Parameter verlangt (*default constructor*). In diesem Falle wird *root* einfach auf 0 gesetzt werden.

TestFriends.C

```
  ListOfFriends list2(list1);
```

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

TestFriends.C

```
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.