- Am Beispiel der *Array*-Klasse wurde bereits das RAII-Prinzip demonstriert.
- Dies lässt sich auch auf rekursive Datenstrukturen übertragen.
- Das folgende Beispiel zeigt dies für einen einfachen, sortierten Binärbaum.
- Der Einfachheit halber bleibt in dem Beispiel die gesamte rekursive Datenstruktur privat, weil das die Verantwortung des Aufräumens erleichtert.
- Ansonsten werden *smart pointers* benötigt, die noch später in der Vorlesung vorgestellt werden.

ListOfFriends.hpp

```
class ListOfFriends {
   public:
      // constructor
      ListOfFriends();
      ListOfFriends(const ListOfFriends& other);
      ListOfFriends(ListOfFriends&& other);
      ~ListOfFriends();
      friend void swap(ListOfFriends& l1, ListOfFriends& l2);

      // assignment
      ListOfFriends& operator=(ListOfFriends other);

      // printing
      void print();

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

   private:
      struct Node* root;
      void addto(Node*& p, Node* newNode);
      void visit(const Node* 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* 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* 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⟩

ListOfFriends.cpp

```
Node::Node(const Node* 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
```

- 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.

⟨delete-expression⟩ $\longrightarrow$ [„::“] **delete** ⟨cast-expression⟩

$\longrightarrow$ [„::“] **delete** „[“ „]“ ⟨cast-expression⟩

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

```
void swap(ListOfFriends& l1, ListOfFriends& l2) {
   std::swap(l1.root, l2.root);
}

ListOfFriends::ListOfFriends(ListOfFriends&& other) : ListOfFriends() {
   swap(*this, other);
} // ListOfFriends::ListOfFriends
```

- Der Übernahmekonstruktor (*move constructor*) wird ähnlich wie beim kopierenden Konstruktor implizit aufgerufen.
- Entsprechend der *copy-and-swap*-Vorgehensweise wird dieser auf *swap* zurückgeführt.
- Dies stellt sicher, dass das Quellobjekt in einem Zustand hinterlassen wird, das einen Abbau durch den Dekonstruktor zulässt.

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=(ListOfFriends other) {
   swap(*this, other);
   return *this;
} // ListOfFriends::operator=
```

- Da der voreingestellte Zuweisungs-Operator nur den Wurzelzeiger kopieren würde, muss einer explizit definiert werden.
- Der Parameter wird hier per *call-by-value* übermittelt. Je nach Kontext kommt hier der normale Kopierkonstruktor oder der Verschiebekonstruktor zum Einsatz. Auf der lokalen Kopie ist dann die *swap*-Operation zulässig.

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* p) {
   if (p) {
      visit(p->left);
      std::cout << p->f.get_name() << ": " <<
         p->f.get_info() << std::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. Bei der Parameterübergabe wird der Parameter kopierkonstruiert von *list1*, wonach per *swap* die Zeiger ausgetauscht werden.

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*).
- Der Rückgabewert der Funktion *gen_friends* ist ein temporäres Objekt. Wenn dies an *list4* zugewiesen wird, wird der Parameter mit dem Verschiebekonstruktor erzeugt und dann per *swap* die Zeiger ausgetauscht. Die Datenstruktur wird nirgends dupliziert.
- Danach kommt es zur Ausführung des Zuweisungs-Operators, der den Baum von *list1* dupliziert und bei *list3* einhängt.