- 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* 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⟩

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
```

- 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* const 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.

Function.hpp

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual const std::string& get_name() const = 0;
      virtual double execute(double x) const = 0;
}; // class Function
```

- Polymorphe Methoden einer Basis-Klasse können in einer abgeleiteten Klasse überdefiniert werden.
- Eine Methode wird durch das Schlüsselwort **virtual** als polymorph gekennzeichnet.
- Dies wird auch als *dynamischer Polymorphismus* bezeichnet, da die auszuführende Methode zur Laufzeit bestimmt wird,

Function.hpp

```
virtual const std::string& get_name() const = 0;
```

- Die Angabe von $= 0$ am Ende einer Signatur einer polymorphen Methode ermöglicht den Verzicht auf eine zugehörige Implementierung.
- In diesem Falle gibt es nur Implementierungen in abgeleiteten Klassen und nicht in der Basis-Klasse.
- So gekennzeichnete Methoden werden *abstrakt* genannt.
- Klassen mit mindestens einer solchen Methode werden *abstrakte Klassen* genannt.
- Abstrakte Klassen können nicht instantiiert werden.

Function.hpp

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual const std::string& get_name() const = 0;
      virtual double execute(double x) const = 0;
}; // class Function
```

- Wenn wie in diesem Beispiel alle Methoden abstrakt sind (oder wie beim Dekonstruktor innerhalb der Klassendeklaration implementiert werden), kann die zugehörige Implementierung vollständig entfallen. Entsprechend gibt es keine zugehörige Datei namens *Function.cpp*.
- Implizit definierte Destruktoren und Operatoren müssen explizit als abstrakte Methoden deklariert werden, wenn die Möglichkeit erhalten bleiben soll, sie in abgeleiteten Klassen überzudefinieren.

Sinus.hpp

```
#include <string>
#include "Function.hpp"

class Sinus: public Function {
   public:
      virtual const std::string& get_name() const;
      virtual double execute(double x) const;
}; // class Sinus
```

- *Sinus* ist eine von *Function* abgeleitete Klasse.
- Das Schlüsselwort **public** bei der Ableitung macht diese Beziehung öffentlich. Alternativ wäre auch **private** zulässig. Dies ist aber nur in seltenen Fällen sinnvoll.
- Die Wiederholung des Schlüsselworts **virtual** bei den Methoden ist nicht zwingend notwendig, erhöht aber die Lesbarkeit.
- Da $= 0$ nirgends mehr innerhalb der Klasse *Sinus* verwendet wird, ist die Klasse nicht abstrakt und somit ist eine Instantiierung zulässig.

Sinus.cpp

```
#include <cmath>
#include "Sinus.hpp"

static std::string name {"sin"};

const std::string& Sinus::get_name() const {
   return name;
} // Sinus::get_name

double Sinus::execute(double x) const {
   return std::sin(x);
} // Sinus::execute
```

- Alle Methoden, die nicht abstrakt sind und nicht in einer der Basisklassen definiert worden sind, müssen implementiert werden.
- Hier wird auf die Definition eines Dekonstruktors verzichtet. Stattdessen kommt der leere Dekonstruktor der Basisklasse zum Zuge.

TestSinus.cpp

```
#include <iostream>
#include "Sinus.hpp"

using namespace std;

int main() {
   Function* f(new Sinus());
   double x;

   while (cout << f->get_name() << ": " &&
          cin >> x) {
      cout << f->execute(x) << endl;
   }
   return 0;
} // main
```

- Variablen des Typs *Function* können nicht deklariert werden, weil *Function* eine abstrakte Klasse ist.
- Stattdessen ist es aber zulässig, Zeiger oder Referenzen auf *Function* zu deklarieren, also *Function*∗ oder *Function*&.

TestSinus.cpp

```
#include <iostream>
#include "Sinus.hpp"

using namespace std;

int main() {
   Function* f(new Sinus());
   double x;

   while (cout << f->get_name() << ": " &&
          cin >> x) {
      cout << f->execute(x) << endl;
   }
   return 0;
} // main
```

- Zeiger auf Instantiierungen abgeleiteter Klassen (wie etwa hier das Resultat von **new** *Sinus*()) können an Zeiger der Basisklasse (hier: *Function∗ f*) zugewiesen werden.
- Umgekehrt gilt dies jedoch nicht!

TestSinus.cpp

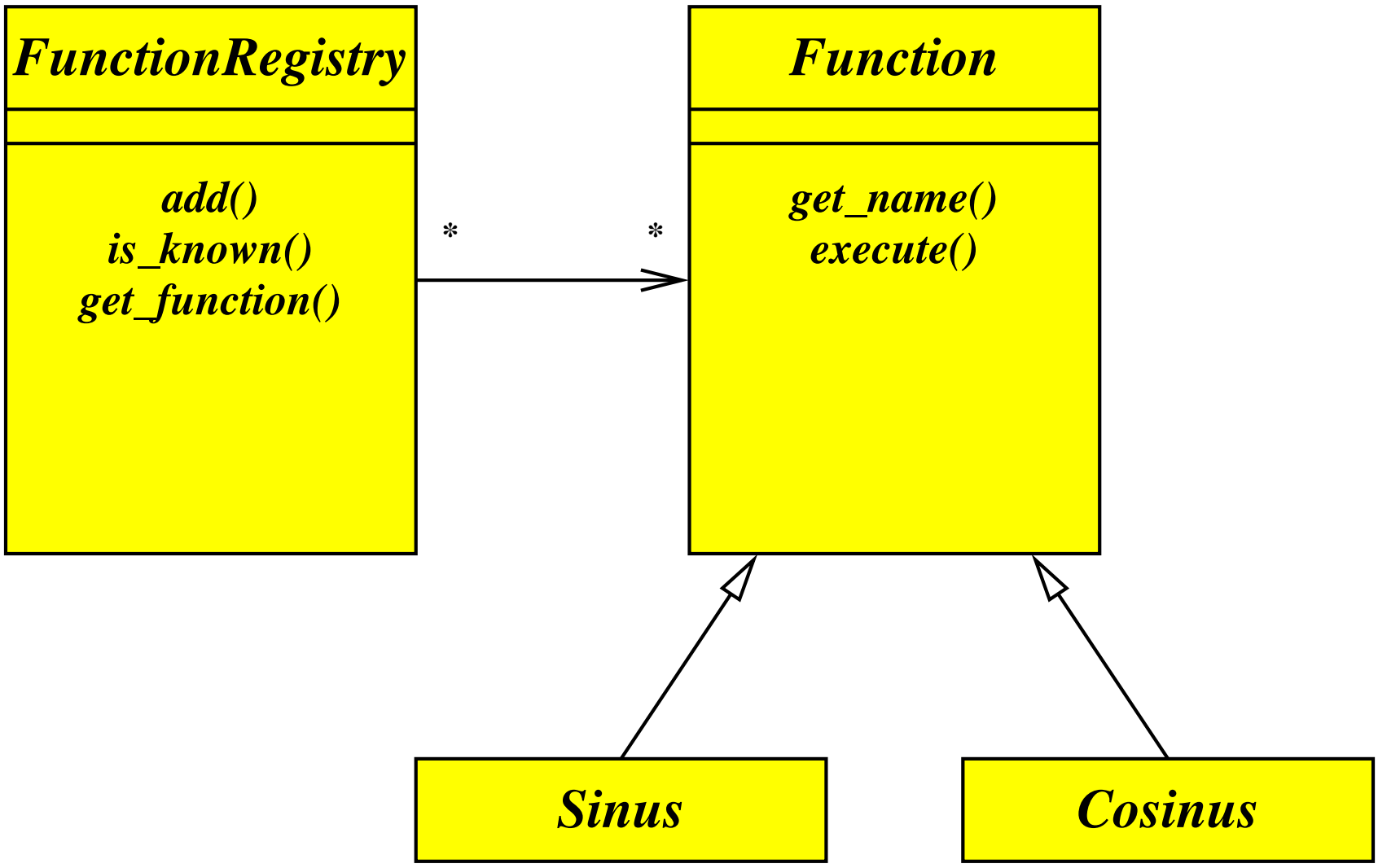
```
#include <iostream>
#include "Sinus.hpp"

using namespace std;

int main() {
   Function* f(new Sinus());
   double x;

   while (cout << f->get_name() << ": " &&
          cin >> x) {
      cout << f->execute(x) << endl;
   }
   return 0;
} // main
```

- Wenn eine Methode mit dem Schlüsselwort **virtual** versehen ist, dann erfolgt die Bestimmung der zugeordneten Methodenimplementierung *erst zur Laufzeit* in Abhängigkeit des dynamischen Typs, der bei Zeigern und Referenzen eine beliebige Erweiterung des deklarierten Typs sein kann.

TestSinus.cpp

```
Function* f(new Sinus());
```

- Fehlt das Schlüsselwort **virtual**, so steht bereits zur Übersetzzeit fest, welche Implementierung aufzurufen ist.
- In diesem Beispiel hat die Variable f den statischen Typ *Function*$*$, während zur Laufzeit hier der dynamische Typ *Sinus*$*$ ist.


FunctionRegistry
add()
is_known()
get_function()
*
*
Function
get_name()
execute()
Sinus
Cosinus

- Die Einführung einer Klasse *FunctionRegistry* erlaubt es, Funktionen über ihren Namen auszuwählen.
- Hiermit ist es beispielsweise möglich, den Namen einer Funktion einzulesen und dann mit dem gegebenen Namen ein zugehöriges Funktionsobjekt zu erhalten.
- Dank der Kompatibilität einer abgeleiteten Klasse zu den Basisklassen ist es möglich, heterogene Listen (d.h. Listen mit Objekten unterschiedlicher Typen) zu verwalten, sofern eine gemeinsame Basisklasse zur Verfügung steht. In diesem Beispiel ist das *Function*.

FunctionRegistry.hpp

```
#include <map>
#include <string>
#include "Function.hpp"

class FunctionRegistry {
   public:
      void add(Function* f);
      bool is_known(const std::string& fname) const;
      Function* get_function(const std::string& fname);
   private:
      std::map< std::string, Function* > registry;
}; // class FunctionRegistry
```

- *map* ist eine Implementierung für assoziative Arrays und gehört zu den generischen Klassen der Standard-Template-Library (STL)
- *map* erwartet zwei Typen als Parameter: den Index- und den Element-Typ.
- Hier werden Zeichenketten als Indizes verwendet (Datentyp *string*) und die Elemente sind Zeiger auf Funktionen (Datentyp *Function*$*$).

- Generell können heterogene Datenstrukturen nur Zeiger oder Referenzen auf den polymorphen Basistyp aufnehmen, da
  - abstrakte Klassen nicht instantiiert werden können und
  - das Kopieren eines Objekts einer erweiterten Klasse zu einem Objekt der Basisklasse (falls überhaupt zulässig) die Erweiterungen ignorieren würde. Dies wird im Englischen *slicing* genannt. (In Oberon nannte dies Wirth eine Projektion.)

FunctionRegistry.cpp

```
#include <string>
#include "FunctionRegistry.hpp"

void FunctionRegistry::add(Function* f) {
   registry[f->get_name()] = f;
} // FunctionRegistry::add

bool FunctionRegistry::is_known(const std::string& fname) const {
   return registry.find(fname) != registry.end();
} // FunctionRegistry::is_known

Function* FunctionRegistry::get_function(const std::string& fname) {
   return registry[fname];
} // FunctionRegistry::get_function
```

- Instantiierungen der generischen Klasse *map* können analog zu regulären Arrays verwendet werden, da der []-Operator für sie überladen wurde.
- *registry.find* liefert einen Iterator, der auf *registry.end*() verweist, falls der gegebene Index bislang noch nicht belegt wurde.

FunctionRegistry.cpp

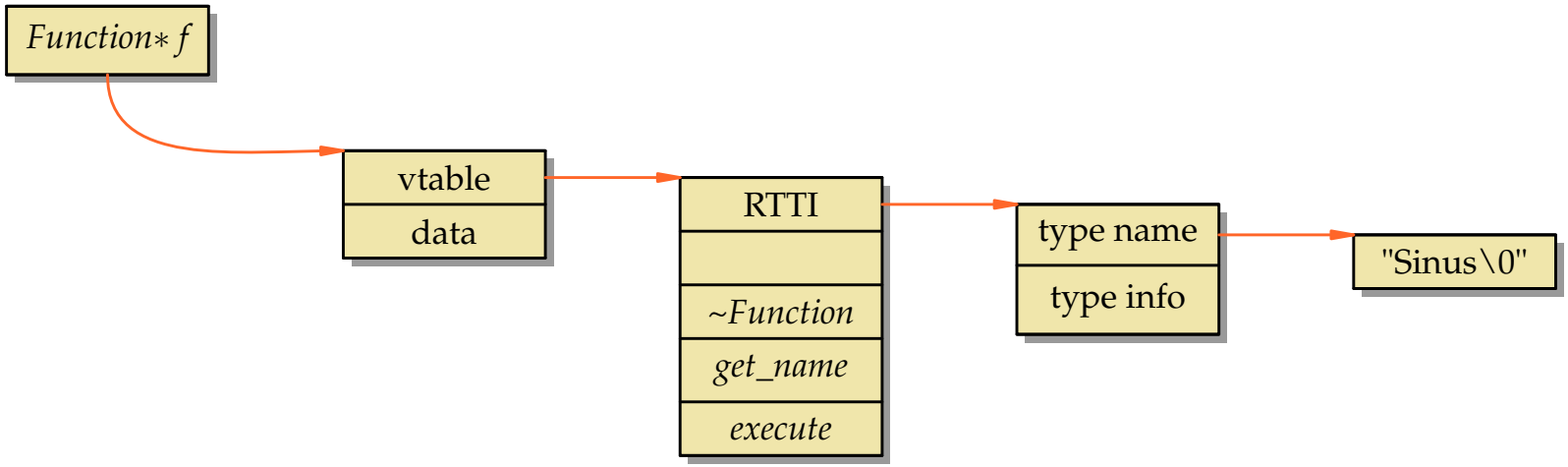
```
bool FunctionRegistry::is_known(const std::string& fname) const {
   return registry.find(fname) != registry.end();
} // FunctionRegistry::is_known
```

- Die STL-Container-Klassen wie *map* arbeiten mit Iteratoren.
- Iteratoren werden weitgehend wie Zeiger behandelt, d.h. sie können dereferenziert werden und vorwärts oder rückwärts zum nächsten oder vorherigen Element gerückt werden.
- Die *find*-Methode liefert nicht unmittelbar das gewünschte Objekt, sondern einen Iterator, der darauf zeigt.
- Die *end*-Methode liefert einen Iterator-Wert, der für das Ende steht.
- Durch einen Vergleich kann dann festgestellt werden, ob das gewünschte Objekt gefunden wurde.

TestFunctions.cpp

```
#include <iostream>
#include "Sinus.hpp"
#include "Cosinus.hpp"
#include "FunctionRegistry.hpp"

using namespace std;

int main() {
   FunctionRegistry registry;
   registry.add(new Sinus());
   registry.add(new Cosinus());

   string fname; double x;
   while (cout << ": " &&
          cin >> fname >> x) {
      if (registry.is_known(fname)) {
         Function* f = registry.get_function(fname);
         cout << f->execute(x) << endl;
      } else {
         cout << "Unknown function name: " << fname << endl;
      }
   }
} // main
```

Wie teuer ist dynamischer Polymorphismus? 164



- Nicht-polymorphe Methoden und reguläre Funktionen können in C++ direkt aufgerufen werden, d.h. die Sprungadresse ist direkt im Maschinen-Code verankert.
- Bei polymorphen Methoden muss zunächst hinter dem Objektzeiger der sogenannte *vtable*-Zeiger geladen werden, hinter dem sich wiederum eine Liste mit Funktionszeigern zu den einzelnen Methoden verbirgt.
- Die Kosten einer polymorphen Methode belaufen sich entsprechend auf zwei nicht parallelisierbare Speicherzugriffe. Im ungünstigsten Falle (d.h. nichts davon ist im Cache) kostet dies bei aktuellen Systemen zwischen 140 und 200 ns.

TestFunctions.s

```
        call    _ZN16FunctionRegistry12get_functionERKSs
        addl    $12, %esp
        movl    (%eax), %edx
        pushl   -60(%ebp)
        pushl   -64(%ebp)
        pushl   %eax
        call    *12(%edx)
```

- Dies ist der optimierte Assembler-Text für die x86-Architecktur, der für $f{->}execute(x)$ nach dem Aufruf von $f = registry.get\_function(fname)$ generiert wird.
- Das Resultat von *get_function* liegt nach dem Aufruf in *%eax*. Dieser Zeiger wird dann dereferenziert und der dann zur Verfügung stehende *vtable*-Zeiger in *%edx* abgelegt. Das ist der erste Speicherzugriff.
- Danach werden die beiden Parameter f und x jeweils mit *pushl* auf den Stack gelegt.
- Schließlich wird der *vtable*-Zeiger dereferenziert (zweiter Speicherzugriff), um den Zeiger auf die aufzurufende Methode zu holen und diesen aufzurufen.

- Dynamischer Polymorphismus verschenkt u.U. Optimierungspotential.
- Wenn eine Klassen-Implementierung bei der Übersetzung des Klienten zugänglich ist (z.B. weil sie sich in der entsprechenden Header-Datei befindet), ergibt sich für den Übersetzer die Möglichkeit, den Methodenaufruf wegzuoptimieren und den ⟨function-body⟩ der Methode direkt an der Stelle des Aufrufs hineinzugenerieren.
- Mit dem Schlüsselwort **inline** kann auch direkt darum gebeten werden.
- All dies entfällt bei dynamischen Polymorphismus, weil erst zur Laufzeit die zugehörige Methode ermittelt wird.

- Da der Aufruf polymorpher Methoden (also solcher Methoden, die mit **virtual** ausgezeichnet sind) zusätzliche Kosten während der Laufzeit verursacht, stellt sich die Frage, wann dieser Aufwand gerechtfertigt ist.
- Sinnvoll ist dynamischer Polymorphismus insbesondere, wenn
  - Container mit Zeiger oder Referenzen auf heterogene Objekte gefüllt werden, die alle eine Basisklasse gemeinsam haben oder
  - unbekannte Erweiterungen einer Basisklasse erst zur Laufzeit geladen werden.
- Wenn sich zur Übersetzzeit bereits ermitteln lässt, welche Methoden aufzurufen sind, dann lässt sich das in C++ auf Basis des statischen Polymorphismus besser umsetzen.

```
Sinus* sf = dynamic_cast<Sinus*>(f);
if (sf) {
   cout << "appeared to be sin" << endl;
} else {
   cout << "appeared to be something else" << endl;
}
```

- Typ-Konvertierungen von Zeigern bzw. Referenzen abgeleiteter Klassen in Richtung zu Basisklassen ist problemlos möglich. Dazu wird kein besonderer Operator benötigt.
- In der umgekehrten Richtung kann eine Typ-Konvertierung mit Hilfe des **dynamic_cast**-Operators versucht werden.
- Diese Konvertierung ist erfolgreich, wenn es sich um einen Zeiger oder Referenz des gegebenen Typs handelt (oder eine Erweiterung davon).
- Im Falle eines Misserfolgs liefert **dynamic_cast** einen Nullzeiger.

```
#include <typeinfo>
// ...
const std::type_info& ti{typeid(*f)};
cout << "type of f = " << ti.name() << endl;
```

- Seit C++11 gibt es im Rahmen des Standards *first-class*-Objekte für Typen.
- Der **typeid**-Operator liefert für einen Ausdruck oder einen Typen ein Typobjekt vom Typ *std*::**type_info**.
- *std*::**type_info** kann als Index für diverse Container-Klassen benutzt werden und es ist auch möglich, den Namen abzufragen.
- Wie der Name aber tatsächlich aussieht, ist der Implementierung überlassen. Dies muss nicht mit dem Klassennamen übereinstimmen.

- Die bisher zu C++ erschienen ISO-Standards (bis einschließlich C++14) sehen das dynamische Laden von Klassen nicht vor.
- Der POSIX-Standard (IEEE Standard 1003.1) schließt einige C-Funktionen ein, die das dynamische Nachladen von speziell übersetzten Modulen (*shared objects*) ermöglichen.
- Diese Schnittstelle kann auch von C++ aus genutzt werden, da grundsätzlich C-Funktionen auch von C++ aus verwendbar sind.
- Es sind hierbei allerdings Feinheiten zu beachten, da wegen des Überladens in C++ Symbolnamen auf der Ebene des Laders nicht mehr mit den in C++ verwendeten Namen übereinstimmen. Erschwerend kommt hinzu, dass die Abbildung von Namen in C++ in Symbolnamen – das sogenannte *name mangling* – nicht standardisiert ist.

```
#include <dlfcn.h>
#include <link.h>

void* dlopen(const char* pathname, int mode);
char* dlerror(void);
```

- *dlopen* lädt ein Modul (*shared object*, typischerweise mit der Dateiendung „.so“), dessen Dateiname bei *pathname* spezifiziert wird.
- Der Parameter *mode* legt zwei Punkte unabhängig voneinander fest:
  - Wann werden die Symbole aufgelöst? Entweder sofort (*RTLD_NOW*) oder so spät wie möglich (*RTLD_LAZY*). Letzteres wird normalerweise bevorzugt.
  - Sind die geladenen globalen Symbole für später zu ladende Module sichtbar (*RTLD_GLOBAL*) oder wird ihre Sichtbarkeit lokal begrenzt (*RTLD_LOCAL*)? Hier wird zur Vermeidung von Konflikten typischerweise *RTLD_LOCAL* gewählt.
- Wenn das Laden nicht klappt, dann kann *dlerror* aufgerufen werden, um eine passende Fehlermeldung abzurufen.

```
#include <dlfcn.h>

void* dlsym(void* restrict handle, const char* restrict name);
int dlclose(void* handle);
```

- Die Funktion *dlsym* erlaubt es, Symbolnamen in Adressen zu konvertieren. Im Falle von Funktionen lässt sich auf diese Weise ein Funktionszeiger gewinnen. Zu beachten ist hier, dass nur bei C-Funktionen davon ausgegangen werden kann, dass der C-Funktionsname dem Symbolnamen entspricht. Bei C++ ist das ausgeschlossen. Als *handle* wird der **return**-Wert von *dlopen* verwendet, *name* ist der Symbolname.
- Mit *dlclose* kann ein nicht mehr benötigtes Modul wieder entfernt werden.

```
extern "C" void do_something() {
   // beliebiger C++-Programmtext
}
```

- In C++ kann eine Funktion mit **extern** "C" ausgezeichnet werden.
- Diese Funktion ist dann von C aus unter ihrem Namen aufrufbar.
- Ein Überladen solcher Funktionen ist naturgemäß nicht möglich, da C dies nicht unterstützt.
- Innerhalb dieser Funktion sind allerdings beliebige C++-Konstrukte möglich.
- Ein solche C-Funktion kann benutzt werden, um ein Objekt der C++-Klasse zu konstruieren oder ein Objekt einer passenden Factory-Klasse zu erzeugen, mit der Objekte der eigentlichen Klasse konstruiert werden können.

Sinus.cpp

```
extern "C" Function* construct() {
   return new Sinus();
}
```

- Im Falle sogenannter Singleton-Objekte (d.h. Fälle, bei denen typischerweise pro Klasse nur ein Objekt erzeugt wird), genügt eine einfache Konstruktor-Funktion.
- Diese darf sogar einen global nicht eindeutigen Namen tragen – vorausgesetzt, wir laden das Modul mit der Option *RTLD_LOCAL*. Dann ist das entsprechende Symbol nur über den von *dlopen* zurückgelieferten Zeiger in Verbindung mit der *dlsym*-Funktion zugänglich.

DynFunctionRegistry.hpp

```
class DynFunctionRegistry {
   public:
      // constructors
      DynFunctionRegistry();
      DynFunctionRegistry(const std::string& dirname);

      void add(Function* f);
      bool is_known(const std::string& fname);
      Function* get_function(const std::string& fname);
   private:
      const std::string dir;
      std::map< std::string, Function* > registry;
      Function* dynload(const std::string& fname);
}; // class DynFunctionRegistry
```

- Neben dem Default-Konstruktor gibt es jetzt einen weiteren, der einen Verzeichnisnamen erhält, in dem die zu ladenden Module gesucht werden.
- Ferner kommt noch die private Methode *dynload* hinzu, deren Aufgabe es ist, ein Modul, das die angegebene Funktion implementiert, dynamisch nachzuladen und ein entsprechendes Singleton-Objekt zu erzeugen.

DynFunctionRegistry.cpp

```
typedef Function* FunctionConstructor();

Function* DynFunctionRegistry::dynload(const std::string& name) {
   std::string path = dir;
   if (path.size() > 0) path += "/";
   path += name; path += ".so";
   void* handle = dlopen(path.c_str(), RTLD_LAZY | RTLD_LOCAL);
   if (!handle) return 0;
   FunctionConstructor* constructor =
      (FunctionConstructor*) dlsym(handle, "construct");
   if (!constructor) {
      dlclose(handle); return 0;
   }
   return constructor();
}
```

- Zunächst wird aus *name* ein Pfad bestimmt, unter der das passende Modul abgelegt sein könnte.
- Dann wird mit *dlopen* versucht, es zu laden.
- Wenn dies erfolgreich war, wird mit Hilfe von *dlsym* die Adresse der *construct*-Funktion ermittelt und diese im Erfolgsfalle aufgerufen.

Abrufen einer Funktion

DynFunctionRegistry.cpp

```
Function* DynFunctionRegistry::get_function(const std::string& fname) {
   auto it = registry.find(fname);
   Function* f;
   if (it == registry.end()) {
      f = dynload(fname);
      if (f) {
         add(f);
         if (f->get_name() != fname) registry[fname] = f;
      }
   } else {
      f = it->second;
   }
   return f;
} // FunctionRegistry::get_function
```

- Innerhalb der *map*-Template-Klasse gibt es ebenfalls einen *iterator*-Typ, der hier mit dem Resultat von *find* initialisiert wird.
- Wenn dieser Iterator dereferenziert wird, liefert ein Paar mit den Komponenten *first* (Index) und *second* (eigentlicher Wert hinter dem Index).
- Falls der Name bislang nicht eingetragen ist, wird mit Hilfe von *dynload* versucht, das zugehörige Modul dynamisch nachzuladen.

DynFunctionRegistry.cpp

```
auto it = registry.find(fname);
```

- Beginnend mit C++11 kann bei einer Deklaration auf die Spezifikation eines Typs mit Hilfe des Schlüsselworts **auto** verzichtet werden, wenn sich der gewünschte Typ von der Initialisierung ableiten lässt.
- In diesem Beispiel muss nicht der lange Typname *std*::*map*< *std*::*string*, *Function*∗ >::*iterator* hingeschrieben werden, weil der Übersetzer das selbst automatisiert von dem Rückgabetyp von *registry*.*find*() ableiten kann.