Function.h

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual 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.

Function.h

```
virtual 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.h

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual 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.C*.
- Implizite Konstruktoren, Destruktoren und Operatoren müssen explizit als abstrakte Methoden deklariert werden, wenn die Möglichkeit erhalten bleiben soll, sie in abgeleiteten Klassen überzudefinieren.

Sinus.h

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

class Sinus: public Function {
   public:
      virtual 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.C

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

std::string Sinus::get_name() const {
   return "sin";
} // 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.C

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

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.C

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

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.C

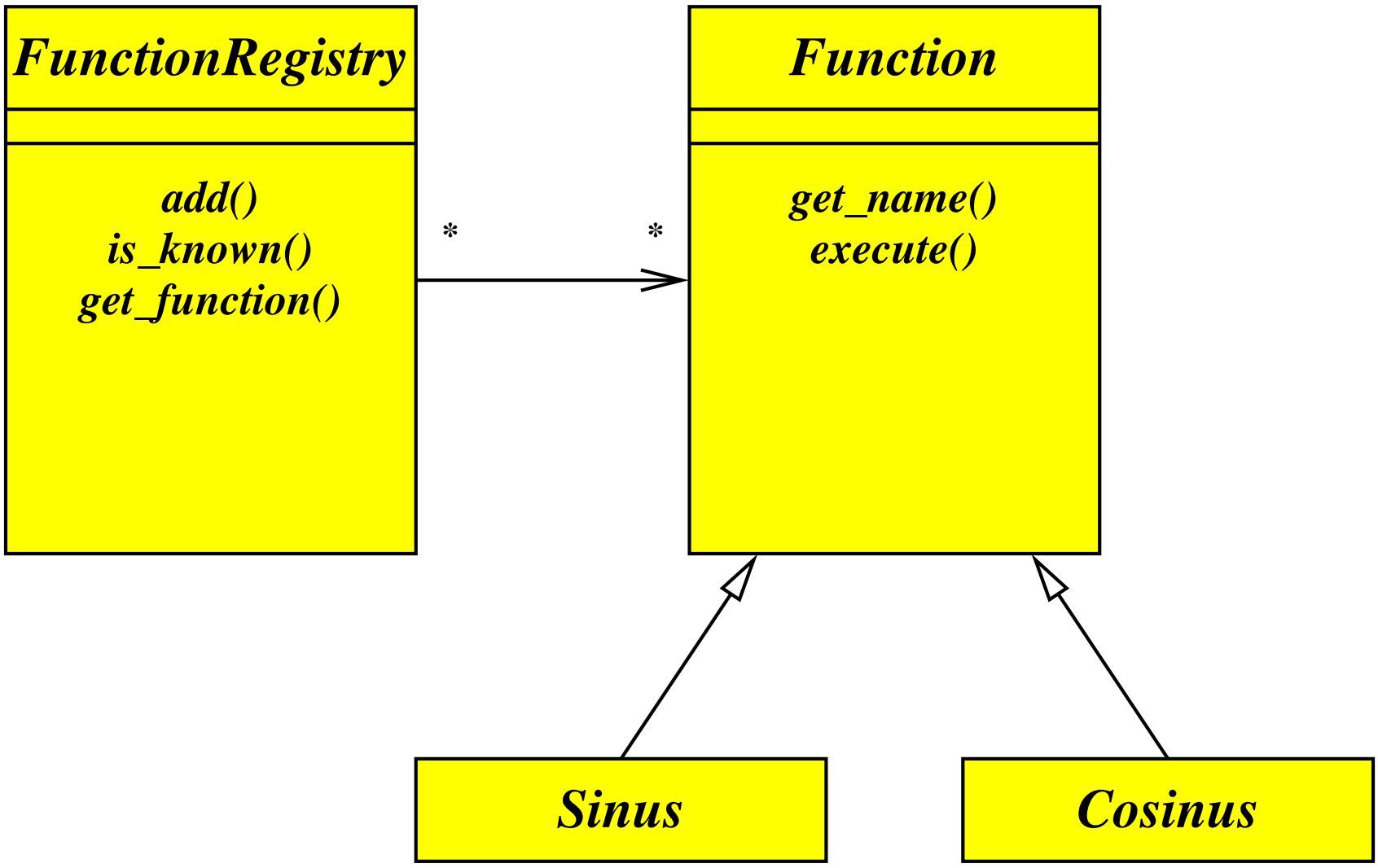
```
#include <iostream>
#include "Sinus.h"

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 vom dynamischen Typ, der bei Zeigern und Referenzen eine beliebige Erweiterung des deklarierten Typs sein kann.

TestSinus.C

```
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 der dynamische Typ hier *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.h

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

class FunctionRegistry {
   public:
      void add(Function* f);
      bool is_known(std::string fname) const;
      Function* get_function(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 Typ der Indizes 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.C

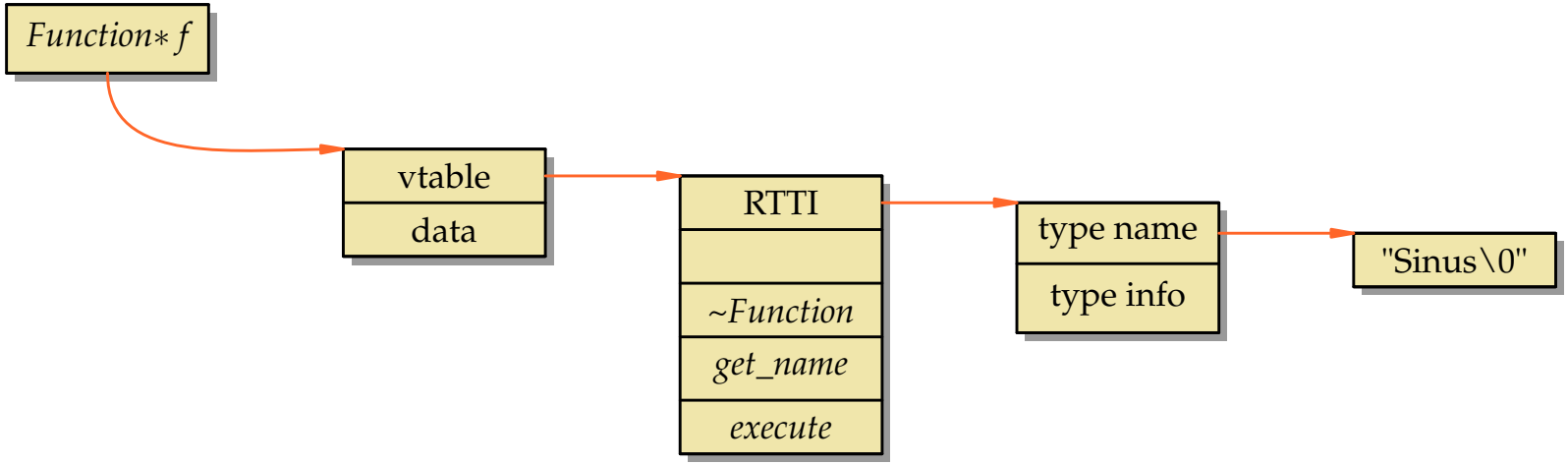
```
#include <string>
#include "FunctionRegistry.h"

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

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

Function* FunctionRegistry::get_function(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.

TestFunctions.C

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

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? 185



- 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 ca. 200 ns.

- 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.

- Die bisher zu C++ erschienen ISO-Standards (bis einschließlich ISO 14882-2003) 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.C

```
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.h

```
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.C

```
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 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.C

```
Function* DynFunctionRegistry::get_function(const std::string& fname) {
  std::map< std::string, Function* >::iterator 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.