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.

Die Benutzung einer polymorphen Klasse

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*&.

Die Benutzung einer polymorphen Klasse 50

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

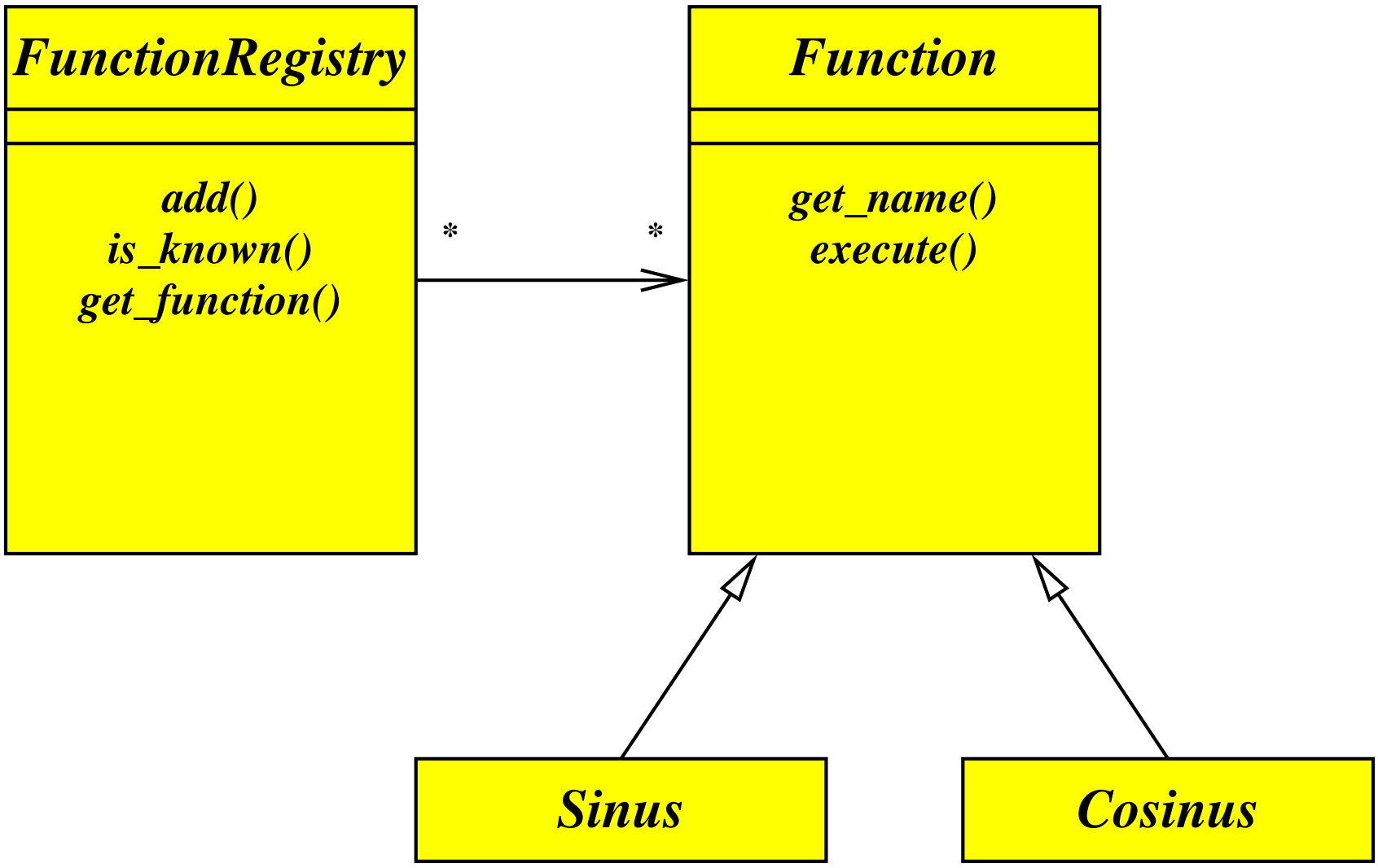
```
#include <iostream>
#include "Sinus.hpp"

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
```