

counter.hpp

```
class Counter {
public:
    // constructors
    Counter() : counter{0} {
    }
    Counter(int counter) : counter{counter} {
    }
    // accessors
    int get() const {
        return counter;
    }
    // mutators
    int increment() {
        assert(counter < INT_MAX);
        return ++counter;
    }
    int decrement() {
        assert(counter > INT_MIN);
        return --counter;
    }
private:
    int counter;
};
```

counter.hpp

```
private:  
    int counter;
```

- Datenfelder sollten normalerweise privat gehalten werden, um den direkten Zugriff darauf zu verhindern. Stattdessen ist es üblich, entsprechende Zugriffsmethoden (*accessors*, *mutators*) zu definieren.

`counter.hpp`

```
Counter() : counter{0} {  
}  
Counter(int counter) : counter{counter} {  
}
```

- Eine Klasse kann beliebig viele Konstruktoren anbieten, solange sie sich in ihrer Signatur unterscheiden.
- Wenn kein Konstruktor angegeben ist, generiert der Übersetzer automatisch einen parameterlosen Konstruktor, der, sofern möglich, sämtliche Datenfelder analog ohne Parameter konstruiert.
- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.

$\langle \text{ctor-initializer} \rangle$	\longrightarrow	„:“ $\langle \text{mem-initializer-list} \rangle$
$\langle \text{mem-initializer-list} \rangle$	\longrightarrow	$\langle \text{mem-initializer} \rangle$ [„...“]
	\longrightarrow	$\langle \text{mem-initializer} \rangle$ [„...“] „,“ $\langle \text{mem-initializer-list} \rangle$
$\langle \text{mem-initializer} \rangle$	\longrightarrow	$\langle \text{mem-initializer-id} \rangle$ „(“ [$\langle \text{expression-list} \rangle$] „)“
	\longrightarrow	$\langle \text{mem-initializer-id} \rangle$ $\langle \text{braced-init-list} \rangle$
$\langle \text{mem-initializer-id} \rangle$	\longrightarrow	$\langle \text{class-or-decltype} \rangle$
	\longrightarrow	$\langle \text{identifier} \rangle$
$\langle \text{braced-init-list} \rangle$	\longrightarrow	„{“ $\langle \text{initializer-list} \rangle$ [„,“] „}“
	\longrightarrow	„{“ „}“

- Seit C++11 wird die $\{...\}$ -Notation ($\langle \text{braced-init-list} \rangle$) bevorzugt, da sie einen unschönen grammatikalischen Konflikt vermeidet und mehr Möglichkeiten erlaubt.

- Es ist in C++ sehr wichtig, zwischen einer Initialisierung mit Hilfe eines `<ctor-initializer>` und einer regulären Zuweisung innerhalb des `<compound-statement>` des Konstruktors zu unterscheiden.
- Bevor das `<compound-statement>` des Konstruktors ausgeführt wird, müssen alle Teilobjekte konstruiert sein. Wenn die Initialisierung innerhalb des `<ctor-initializer>` fehlt, dann wird jeweils der parameterlose Konstruktor verwendet.
- Wenn der parameterlose Konstruktor für eines der Teilobjekte nicht zur Verfügung stehen sollte, dann geht es überhaupt nicht ohne die Konstruktion innerhalb des `<ctor-initializer>`.
- Eine implizite automatische Konstruktion in Kombination mit einer späteren Zuweisung kann zu ineffizienteren Code führen. Daher wird grundsätzlich empfohlen, soweit wie möglich alle Teilobjekte innerhalb des `<ctor-initializer>` zu initialisieren.
- Die Reihenfolge im `<ctor-initializer>` sollte der der Deklaration der Teilobjekte entsprechen. In letzterer Reihenfolge werden sie ausgeführt.

`counter.hpp`

```
Counter() : counter{0} {  
}  
Counter(int counter) : counter{counter} {  
}
```

- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.
- Bei der `<mem-initializer-id>` wird nur unter den zu initialisierenden Namen der Objektvariablen gesucht bzw. dem Namen der eigenen Klasse oder den Namen der Basisklassen.
- In der `<expression-list>` bzw. der `<initializer-list>` werden zuerst die Parameternamen durchsucht, bevor die Namen der Objektvariablen in Betracht gezogen werden.

intinit.cpp

```
int main() {
    cout << "Testing..." << endl;
    int i; // undefined
    cout << "i = " << i << endl;
    int j{17}; // well defined
    cout << "j = " << j << endl;
    int k{}; // well defined: 0
    cout << "k = " << k << endl;
}
```

- Die elementare Datentypen bieten ebenfalls parameterlose Konstruktoren an und einen Konstruktor mit einem Parameter des entsprechenden Typs.
- Wenn jedoch kein Konstruktor explizit aufgerufen wird, erfolgt keine Initialisierung.

```
clonmel$ intinit
Testing...
i = 4927
j = 17
k = 0
clonmel$
```

`counter.hpp`

```
int get() const {  
    return counter;  
}
```

- Zugriffsmethoden, die den abstrakten Zustand des Objekts nicht verändern dürfen, werden mit **const** ausgezeichnet.
- Nur diese Methoden dürfen aufgerufen werden, wenn das Objekt in einem Kontext nur lesenderweise zur Verfügung steht.
- Mit **mutable** deklarierte Variablen dürfen auch von **const**-Methoden verändert werden. Dies ist sinnvoll etwa zur Vermeidung sich sonst wiederholender Berechnungen und sollte nicht zu außen sichtbaren Veränderungen führen. (Abstrakter vs. konkreter Zustand eines Objekts.)


```
void f(Counter counter) {
    counter.increment();
    cout << "in f: counter = " << counter.get() << endl;
}

int main() {
    Counter counter{1};
    cout << "before f: " << counter.get() << endl;
    f(counter);
    cout << "after f: " << counter.get() << endl;
}
```

- In C++ werden Parameter grundsätzlich *per call by value* übergeben.
- Dies bedeutet, dass ein neues Objekt konstruiert wird, das ein Klon des als Parameter übergebenen Objekts ist.
- Hierfür wird implizit der Kopierkonstruktor verwendet, der vom Übersetzer automatisiert erstellt wird, sofern nicht explizit einer definiert wird oder dies unterbunden wird.

```
clonmel$ testit
before f: 1
in f: counter = 2
after f: 1
clonmel$
```

```
void f(Counter& counter) {
    counter.increment();
    cout << "in f: counter = " << counter.get() << endl;
}

int main() {
    Counter counter{1};
    cout << "before f: " << counter.get() << endl;
    f(counter);
    cout << "after f: " << counter.get() << endl;
}
```

- C++ unterstützt Referenztypen, die mit Hilfe des „&“ (im `<declarator>`) gekennzeichnet sind.
- Bei formalen Parametern mit Referenztyp erfolgt dann die Parameterübergabe per *call by reference*.

```
clonmel$ testit
before f: 1
in f: counter = 2
after f: 2
clonmel$
```

```
void f(const Counter& counter) {  
    // counter.increment(); /* not OK */  
    cout << "in f: counter = " << counter.get() << endl;  
}
```

- Wenn der Parameterdeklaration noch **const** hinzugefügt wird, dann erfolgt die Übergabe wie zuvor per Referenz, aber die Funktion darf dann dieses Objekt nicht verändern.
- In dieser Situation dürfen nur Methoden aufgerufen werden, die ebenso mit **const** ausgezeichnet sind.
- Diese Art der Parameterübergabetechnik wird gerne verwendet, wenn die aufgerufene Methode das Objekt nicht verändern soll und gleichzeitig ein möglicherweise kostspieliges Klonen unterbunden werden soll.
- Letzteres ist insbesondere bei Containern ein Problem, die sehr umfangreich werden können mit unabsehbaren Kosten bei einem Klon-Vorgang.

```
Counter(const Counter& orig) : counter{orig.counter} {  
}
```

- Wenn einer der Konstruktoren genau einen Parameter mit einem Referenztyp der eigenen Klasse hat, dann handelt es sich dabei um einen expliziten Kopierkonstruktor.
- Normalerweise wird dieser mit **const** versehen.
- Der Kopierkonstruktor wird dann ggf. implizit verwendet bei der Parameterübergabe, bei **return** und einer Zuweisung.
- Wenn der Kopierkonstruktor nicht explizit deklariert und nicht ausdrücklich unterbunden wird, dann erzeugt der Übersetzer einen, wobei jedes Teilobjekt entsprechend kopierkonstruiert wird. Das funktioniert nur, wenn dies für alle Teilobjekte geht.

- Klassen, die selbst Ressourcen verwalten wie etwa dynamisch angelegte Speicherbereiche, benötigen sehr viel Sorgfalt in C++, damit mit der impliziten Verwendung von Konstruktoren und Methoden keine Probleme entstehen.
- Es ist dabei insbesondere sicherzustellen, dass dynamische Datenstrukturen nur ein einziges Mal freigegeben werden. D.h. das unbemerkte Kopieren von Zeigern ist ein Problem.
- Das folgende Beispiel illustriert dies an einer sehr einfachen Klasse, die ein **int**-Array verwaltet.

array.hpp

```
class Array {
public:
    Array() : nof_elements(0), ip(nullptr) {}
    Array(unsigned int nof_elements) : nof_elements{nof_elements},
        ip{new int[nof_elements] {}} {}
    }
    Array(const Array& other) : nof_elements{other.nof_elements},
        ip{new int[nof_elements]} {
        for (unsigned int i = 0; i < nof_elements; ++i) {
            ip[i] = other.ip[i];
        }
    }
    Array(Array&& other) : nof_elements{other.nof_elements},
        ip{other.ip} {
        other.nof_elements = 0; other.ip = nullptr;
    }
    ~Array() { delete[] ip; }
    Array& operator=(const Array& other) = delete;
    Array& operator=(Array&& other) = delete;
    unsigned int size() const { return nof_elements; }
    int& operator()(unsigned int i) {
        assert(i < nof_elements); return ip[i];
    }
    const int& operator()(unsigned int i) const {
        assert(i < nof_elements); return ip[i];
    }
private:
    unsigned int nof_elements; int* ip;
};
```

array.hpp

```
Array(unsigned int nof_elements) : nof_elements{nof_elements},  
    ip{new int[nof_elements] {}} {  
}
```

- Mit dem **new**-Operator kann Speicher dynamisch belegt werden. Neben einem Typnamen kann auch in Array-Notation eine Dimensionierung mit angegeben werden.
- Hier wird ein **int**-Array mit *nof_elements* Elementen angelegt.
- Der **new**-Operator lässt hier auch sogleich die Initialisierung der neuen Speicherfläche hinzu. Da `{}` hier angegeben ist, wird das Array mit Nullen initialisiert. (Ohne diesen expliziten Hinweis würden die **int** uninitialized bleiben.)

array.hpp

```
~Array() {  
    delete[] ip;  
}
```

- Mit dem Operator **delete[]** kann ein Array wieder freigegeben werden.
- Wenn der Zeiger ein **nullptr** sein sollte, stört das nicht.
- Anders als in Java wird diese Funktion garantiert aufgerufen, wenn die Lebenszeit des Objekts beendet ist.

Der Begriff *Resource Acquisition Is Initialization* (RAII) geht auf Bjarne Stroustrup und Andrew Koenig zurück. Folgende Prinzipien sind damit verknüpft:

- ▶ Ressourcen werden mit Objekten fest verknüpft.
- ▶ Bei der Initialisierung des Objekts wird die Ressource akquiriert.
- ▶ Beim Dekonstruieren des Objekts wird die Ressource freigegeben.

Diese Technik vermeidet Fehler und stellt insbesondere sicher, dass auch im Falle einer Ausnahmenbehandlung (*exception handling*) alles sauber abgebaut und damit freigegeben wird.

Da bei C++ Objekte kopiert und nicht ohne weiteres nur Zeiger einander zugewiesen werden, ist bei Klassen, die mit Ressourcen in Verbindung stehen, Vorsicht geboten:

Wenn immer eine Klasse eine Ressource verwaltet, sind folgende spezielle Methoden immer explizit zu definieren bzw. zu deaktivieren, um die unerwünschte implizite Definition zu unterbinden:

- ▶ Kopier-Konstruktor
- ▶ Zuweisungs-Operator
- ▶ Dekonstruktor

array.hpp

```
Array(const Array& other) : nof_elements{other.nof_elements},
    ip{new int[nof_elements]} {
    for (unsigned int i = 0; i < nof_elements; ++i) {
        ip[i] = other.ip[i];
    }
}
```

- Der Kopierkonstruktor hat die Aufgabe, die gesamte Datenstruktur zu duplizieren.
- Hierfür ist das Kopieren nur des Zeigers unzureichend. Denn dann würde die Klon-Semantik verlorengehen und wir hätten das Problem, dass das Array beim Abbau mehrfach freigegeben würde. Die vom Übersetzer zur Verfügung stehende voreingestellte Implementierung dieses Konstruktors wäre somit fatal.
- Hier wird wiederum dynamisch Speicher von der gegebenen Größe angelegt und dann mit Hilfe der **for**-Schleife die Elemente einzeln kopiert. (Wie das effizienter geht, kommt später.)

array.hpp

```
Array(Array&& other) : nof_elements{other.nof_elements},  
    ip{other.ip} {  
    other.nof_elements = 0;  
    other.ip = nullptr;  
}
```

- Der Verschiebekonstruktor (*move constructor*) wird dann verwendet, wenn das Quellobjekt unmittelbar nach dem Aufruf abgebaut wird. Das ist insbesondere bei temporären Objekten der Fall.
- In diesem Fall können wir die dynamische Datenstruktur einfach übernehmen und müssen dann nur sicherstellen, dass beim Abbau des Quellobjekts keine versehentliche Freigabe der Datenstruktur erfolgt.

array.hpp

```
Array& operator=(const Array& other) = delete;  
Array& operator=(Array&& other) = delete;
```

- Der Übersetzer unterstützt auch implizit Zuweisungen. Bei Objekten mit Zeigern ist hier ebenfalls die voreingestellte Implementierung unzureichend.
- Hier wird gezeigt, wie die voreingestellte Implementierung mit Hilfe von **= delete** unterdrückt werden kann, ohne sie durch eine eigene Implementierung zu ersetzen.

array.hpp

```
int& operator()(unsigned int i) {  
    assert(i < nof_elements); return ip[i];  
}  
const int& operator()(unsigned int i) const {  
    assert(i < nof_elements); return ip[i];  
}
```

- Statt traditioneller *set*- und *get*-Methoden kann auch der direkte Zugriff auf ein ansonsten privates Element gegeben werden, indem eine Referenz zurückgegeben wird.
- Das Resultat kann dann sowohl als *lvalue* (links von einer Zuweisung) als auch als *rvalue* (rechts der Zuweisung) verwendet werden.
- Methoden können auch nach einem Operator benannt werden mit Hilfe des Schlüsselworts **operator**. (Hier ist es der Funktionsoperator `()`.)

```
Array a{10};  
a(1) = 77;  
a(2) = a(1) + 10;
```

```
friend void swap(Array& a1, Array& a2) {
    std::swap(a1.nof_elements, a2.nof_elements);
    std::swap(a1.ip, a2.ip);
}
Array(const Array& other) :
    nof_elements{other.nof_elements},
    ip{new int[nof_elements]} {
    for (unsigned int i = 0; i < nof_elements; ++i) {
        ip[i] = other.ip[i];
    }
}
Array(Array&& other) : Array() {
    swap(*this, other);
}
Array& operator=(Array other) {
    swap(*this, other);
    return *this;
}
```

- Wenn die Zuweisung unterstützt werden soll, dann dient ein *swap*-Operator der Vereinfachung.

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;

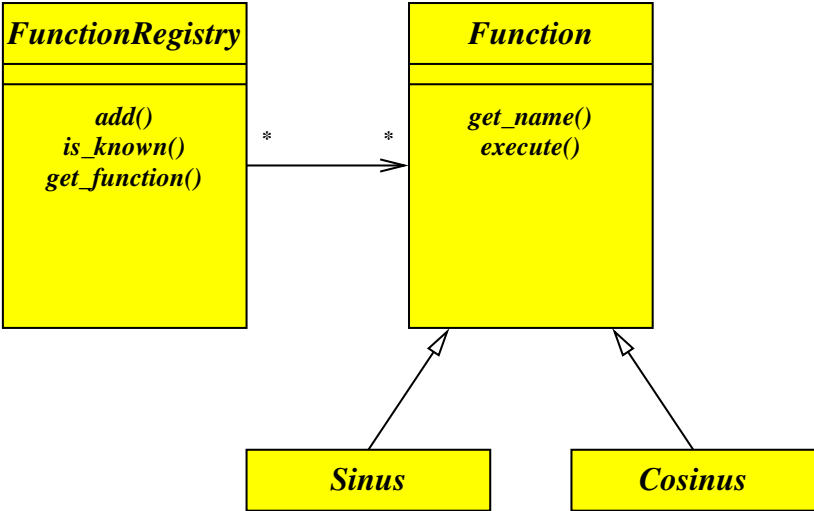
    while (cout << f->get_name() << ": " &&
           cin >> x) {
        cout << f->execute(x) << endl;
    }
    return 0;
} // main
```

- 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.



- 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.

```
#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
```

- 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.