Übungen zu Parallele Programmierung mit C++
Einführung in C++
SS 2015

Andreas F. Borchert

Universität Ulm

24. April 2015

- Bjarne Stroustrup startete sein Projekt *C with Classes* im April 1979 bei den Bell Laboratories nach seinen Erfahrungen mit Simula und BCPL.
- Sein Ziel war es, die Klassen von Simula als Erweiterung zur Programmiersprache C einzuführen, ohne Laufzeiteffizienz zu opfern. Der Übersetzer wurde als Präprozessor zu C implementiert, der *C with Classes* in reguläres C übertrug.
- 1982 begann ein Neuentwurf der Sprache, die dann den Namen C++ erhielt. Im Rahmen des Neuentwurfs kamen virtuelle Funktionen (und damit Polymorphismus), die Überladung von Operatoren, Referenzen, Konstanten und verbesserte Typüberprüfungen hinzu.

- 1985 begann Bell Laboratories mit der Auslieferung von *Cfront*, der C++ in C übersetzte und damit eine Vielzahl von Plattformen unterstützte.
- 1990 wurde für C++ bei ANSI/ISO ein Standardisierungskomitee gegründet.
- Vorschläge für Templates in C++ gab es bereits in den 80er-Jahren und eine erste Implementierung stand 1989 zur Verfügung. Sie wurde 1990 vom Standardisierungskomitee übernommen.
- Analog wurden Ausnahmenbehandlungen 1990 vom Standardisierungskomitee akzeptiert. Erste Implementierungen hierfür gab es ab 1992.
- Namensräume wurden erst 1993 in C++ eingeführt.
- Im September 1998 wurde mit ISO 14882 der erste Standard für C++ veröffentlicht. Die aktuelle Fassung des Standards ist von August 2011 und wird kurz C++11 genannt.

Übersetzungseinheiten in C++ 4

- In C++ sind Übersetzungseinheiten Dateien mit Programmtext, die dem C++-Übersetzer unmittelbar auf der Kommandozeile zum Übersetzen angegeben werden. Als Dateiendung wird hier gerne „.cpp“ benutzt – es sind aber auch viele andere üblich wie etwa „.C“ oder „.cc“.
- Mit Hilfe der **#include**-Direktive des Präprozessors können noch sehr viel mehr Programmtexte indirekt hinzukommen.
- Eine Übersetzungseinheit wird normalerweise direkt in Maschinencode für eine ausgewählte Plattform übersetzt (normalerweise die lokale, ein *cross compiler* kann auch für andere Plattformen übersetzen). Diese Resultate werden auch Objekte genannt (hat nichts mit den OO-Konzepten zu tun).
- Mit Hilfe des *ld* (*linkage editor*) können mehrere Objekte zusammen mit den Bibliotheken zu einem ausführbaren Programm zusammengefügt werden.

⟨translation-unit⟩ ⟶ [⟨declaration-seq⟩]

⟨declaration-seq⟩ ⟶ ⟨declaration⟩

⟶ ⟨declaration-seq⟩ ⟨declaration⟩

⟨declaration⟩ ⟶ ⟨block-declaration⟩

⟶ ⟨function-definition⟩

⟶ ⟨template-declaration⟩

⟶ ⟨explicit-instantiation⟩

⟶ ⟨explicit-specialization⟩

⟶ ⟨linkage-specification⟩

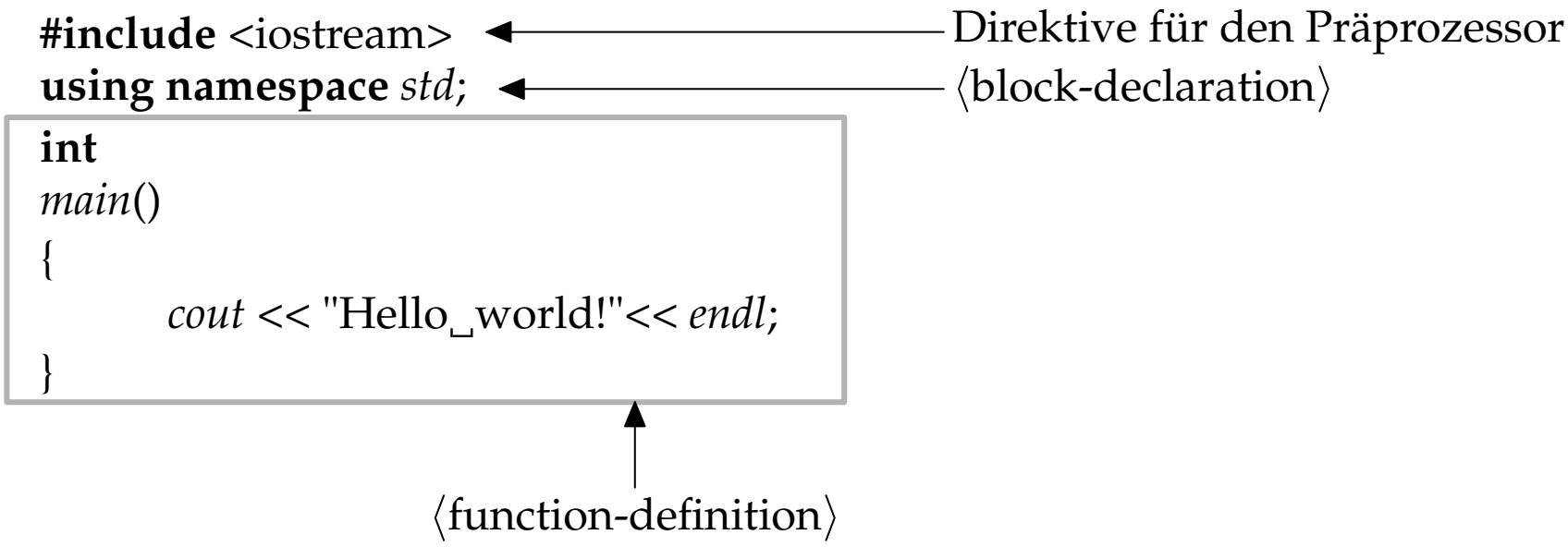
⟶ ⟨namespace-definition⟩

⟶ ⟨empty-declaration⟩

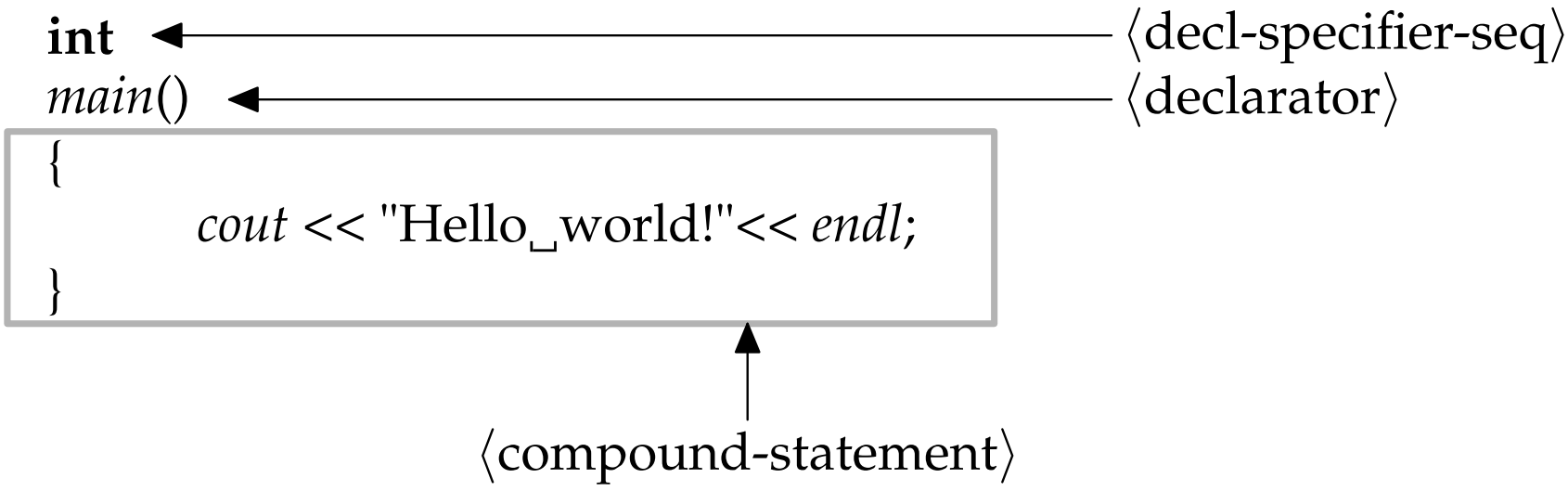
⟶ ⟨attribute-declaration⟩

- Eine Übersetzungseinheit besteht aus einer Sequenz von Deklarationen und Definitionen. Diese darf auch leer sein. (Diese und die folgenden Syntaxspezifikationen wurden dem N3797 entnommen.)

Übersetzungseinheiten in C++ 6



- Präprozessor-Anweisungen betten sich nicht in die C++-Syntax ein. Durch den Präprozessor werden sie durch Text ersetzt, der der C++-Syntax entsprechend sollte. Bei **#include**-Direktiven ist dies der Inhalt der gegebenen Datei (hier *iostream*, die standardmäßig zur Verfügung steht).
- Mit **using namespace** *std* lässt sich alles aus dem *std*-Namensraum ohne Qualifikation verwenden, also etwa *cout* anstelle von *std::cout*.



⟨function-definition⟩ ⟶ [⟨attribute-specifier-seq⟩]
[⟨decl-specifier-seq⟩]
⟨declarator⟩ [⟨virt-specifier-seq⟩]
⟨function-body⟩

⟨function-body⟩ ⟶ [⟨ctor-initializer⟩]
⟨compound-statement⟩

$$
\begin{aligned}
\langle \text{block-declaration} \rangle \quad &\longrightarrow \quad \langle \text{simple-declaration} \rangle \\
&\longrightarrow \quad \langle \text{asm-definition} \rangle \\
&\longrightarrow \quad \langle \text{namespace-alias-definition} \rangle \\
&\longrightarrow \quad \langle \text{using-declaration} \rangle \\
&\longrightarrow \quad \langle \text{using-directive} \rangle \\
&\longrightarrow \quad \langle \text{static\_assert-declaration} \rangle \\
&\longrightarrow \quad \langle \text{alias-declaration} \rangle \\
&\longrightarrow \quad \langle \text{opaque-enum-declaration} \rangle
\end{aligned}
$$

- Blockdeklarationen können in C++ sowohl auf globaler Ebene als auch innerhalb eines Blocks (d.h. inmitten regulären Programmtexts) erfolgen.

⟨simple-declaration⟩ ⟶ [⟨decl-specifier-seq⟩]
[⟨init-declarator-list⟩] „;“
⟶ [⟨attribute-specifier-seq⟩]
[⟨decl-specifier-seq⟩]
⟨init-declarator-list⟩ „;“

- Die Mehrzahl der Deklarationen in C++ fällt unter die Rubrik der ⟨simple-declaration⟩. Dazu gehören u.a. Variablen- und Klassendeklarationen.
- Die wichtigsten Teile einer Deklaration sind die ⟨decl-specifier⟩, die den Grundtyp spezifizieren und der ⟨declarator⟩, der einen Namen mit einer möglicherweise abgeleiteten Variante des Grundtyps assoziiert.
- Bei Klassendeklarationen fällt normalerweise die ⟨init-declarator-list⟩ weg, wenn nicht sofort im Rahmen der Deklaration auch entsprechende Objekte zu instantiieren sind.

⟨decl-specifier-seq⟩ ⟶ ⟨decl-specifier⟩
[⟨attribute-specifier-seq⟩]
⟶ ⟨decl-specifier⟩
⟨decl-specifier-seq⟩
⟨decl-specifier⟩ ⟶ ⟨storage-class-specifier⟩
⟶ ⟨type-specifier⟩
⟶ ⟨function-specifier⟩
⟶ **friend**
⟶ **typedef**
⟶ **constexpr**

⟨type-specifier⟩ ⟶ ⟨trailing-type-specifier⟩
⟶ ⟨class-specifier⟩
⟶ ⟨enum-specifier⟩
⟨class-specifier⟩ ⟶ ⟨class-head⟩
„{“ [⟨member-specification⟩] „}“
⟨class-head⟩ ⟶ ⟨class-key⟩ [⟨attribute-specifier-seq⟩]
[⟨class-head-name⟩ [⟨class-virt-specifier⟩]]
[⟨base-clause⟩]
⟨class-key⟩ ⟶ **class**
⟶ **struct**
⟶ **union**

- Bei **class** sind alle Felder und Methoden per Voreinstellung **private**, bei **struct** sind sie (in Kompatibilität zu C) per Voreinstellung **public**. Bei einer **union** werden sämtliche Datenfelder übereinander gelegt.

Deklarationen innerhalb einer Klasse

⟨member-specification⟩ ⟶ ⟨member-declaration⟩
[⟨member-specification⟩]
⟶ ⟨access-specifier⟩ „:“
[⟨member-specification⟩]

⟨member-declaration⟩ ⟶ [⟨attribute-specifier-seq⟩]
[⟨decl-specifier-seq⟩]
[⟨member-declarator-list⟩]
⟶ ⟨function-definition⟩ [„;“]
⟶ ⟨using-declaration⟩
⟶ ⟨static_assert-declaration⟩
⟶ ⟨template-declaration⟩
⟶ ⟨alias-declaration⟩

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

class Greeting {
   public:
      void hello();
      void hi();
}; // class Greeting

#endif
```

- Klassendeklarationen (mitsamt allen öffentlichen und auch privaten Datenfeldern und Methoden) sind in Dateien, die mit ».hpp«, ».hh« oder ».h« enden, unterzubringen. Hierbei steht ».h« allgemein für eine Header-Datei bzw. ».hh« oder ».hpp« für eine Header-Datei von C++.
- Alle Zeilen, die mit einem # beginnen, enthalten Direktiven für den Makro-Präprozessor. Dieses Relikt aus Assembler- und C-Zeiten ist in C++ erhalten geblieben. Die Konstruktion in diesem Beispiel stellt sicher, dass die Klassendeklaration nicht versehentlich mehrfach in den zu übersetzenden Text eingefügt wird.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Eine Klassendeklaration besteht aus einem Namen und einem Paar geschweifter Klammern, die eine Sequenz von Deklarationen eingrenzen. Die Klassendeklaration wird (wie sonst alle anderen Deklarationen in C++ auch) mit einem Semikolon abgeschlossen.
- Kommentare starten mit »//« und erstrecken sich bis zum Zeilenende.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Die Deklarationen der einzelnen Komponenten einer Klasse, in der C++-Terminologie *member* genannt, fallen in verschiedene Kategorien, die die Zugriffsrechte regeln:

private	nur für die Klasse selbst und ihre Freunde zugänglich
protected	offen für alle davon abgeleiteten Klassen
public	uneingeschränkter Zugang

  Wenn keine der drei Kategorien explizit angegeben wird, dann wird automatisch **private** angenommen.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Alle Funktionen (einschließlich der Methoden einer Klasse) haben einen Typ für ihre Rückgabewerte. Wenn nichts zurückzuliefern ist, dann kann **void** als Typ verwendet werden.
- In Deklarationen folgt jeweils dem Typ eine Liste von durch Kommata getrennten Namen, die mit zusätzlichen Spezifikationen wie etwa () ergänzt werden können.
- Die Angabe () sorgt hier dafür, dass aus *hello* eine Funktion wird, die Werte des Typs **void** zurückliefert, d.h. ohne Rückgabewerte auskommt.

Greeting.cpp

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

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- C++-Programme bzw. die Implementierung einer öffentlichen Schnittstelle in einer Header-Datei werden üblicherweise in separaten Dateien untergebracht.
- Als Dateiendung sind ».cpp«, ».cc« oder ».C« üblich.
- Letztere Variante ist recht kurz, hat jedoch den Nachteil, dass sie sich auf Dateisystemen ohne Unterscheidung von Klein- und Großbuchstaben nicht von der Endung ».c« unterscheiden lässt, die für C vorgesehen ist. (Vorsicht ist hier u.a. bei dem *HFS+*-Dateisystem von Apple geboten.)
- Der Übersetzer erhält als Argumente nur diese Dateien, nicht die Header-Dateien.

Greeting.cpp

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

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- Die Direktive **#include** bittet den Präprozessor um das Einfügen des genannten Textes an diese Stelle in den Eingabetext für den Übersetzer.
- Anzugeben ist ein Dateiname. Wenn dieser in <...> eingeschlossen wird, dann erfolgt die Suche danach nur an Standardplätzen, wozu das aktuelle Verzeichnis normalerweise nicht zählt.
- Wird hingegen der Dateiname in "..." gesetzt, dann beginnt die Suche im aktuellen Verzeichnis, bevor die Standardverzeichnisse hierfür in Betracht gezogen werden.

Greeting.cpp

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

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- Der eigentliche Übersetzer von C++ liest nicht direkt von der Quelle, sondern den Text, den der Präprozessor zuvor generiert hat.
- Andere Texte, die nicht direkt oder indirekt mit Hilfe des Präprozessors eingebunden werden, stehen dem Übersetzer nicht zur Verfügung.
- Entsprechend ist es strikt notwendig, alle notwendigen Deklarationen externer Klassen in Header-Dateien unterzubringen, die dann sowohl bei den Klienten als auch dem implementierenden Programmtext selbst einzubinden sind.

Greeting.cpp

```
void Greeting::hello() {
   std::cout << "Hello, world!" << std::endl;
} // hello()
```

- Methoden werden üblicherweise außerhalb ihrer Klassendeklaration definiert. Zur Verknüpfung der Methode mit der Klasse wird eine Qualifizierung notwendig, bei der der Klassenname und das Symbol :: dem Methodennamen vorangehen. Dies ist notwendig, da prinzipiell mehrere Klassen in eine Übersetzungseinheit integriert werden können.
- Eine Funktionsdefinition besteht aus der Signatur und einem Block. Ein terminierendes Semikolon wird hier nicht verwendet.
- Blöcke schließen eine Sequenz lokaler Deklarationen, Anweisungen und weiterer verschachtelter Blöcke ein.
- Funktionen dürfen nicht ineinander verschachtelt werden.

Greeting.cpp

```
void Greeting::hello() {
   std::cout << "Hello, world!" << std::endl;
} // hello()
```

- Die Präprozessor-Direktive **#include** <iostream> fügte Deklarationen in den zu übersetzenden Text ein, die u.a. auch *cout* innerhalb des Namensraumes *std* deklariert hat. Die Variable *std*::*cout* repräsentiert die Standardausgabe und steht global zur Verfügung.
- Da C++ das Überladen von Operatoren unterstützt, ist es möglich, Operatoren wie etwa << (binäres Verschieben) für bestimmte Typkombinationen zu definieren. Hier wurde die Variante ausgewählt, die als linken Operator einen *ostream* und als rechten Operator eine Zeichenkette erwartet.
- *endl* repräsentiert den Zeilentrenner.
- *cout* << "Hello,␣world!" gibt die Zeichenkette auf *cout* aus, liefert den Ausgabekanal *cout* wieder zurück, wofür der Operator << erneut aufgerufen wird mit der Zeichenkette, die von *endl* repräsentiert wird, so dass der Zeilentrenner ebenfalls ausgegeben wird.

SayHello.cpp

```
#include "Greeting.hpp"

int main() {
   Greeting greeting;
   greeting.hello();
   greeting.hello();
   greeting.hello();
   return 0;
} // main()
```

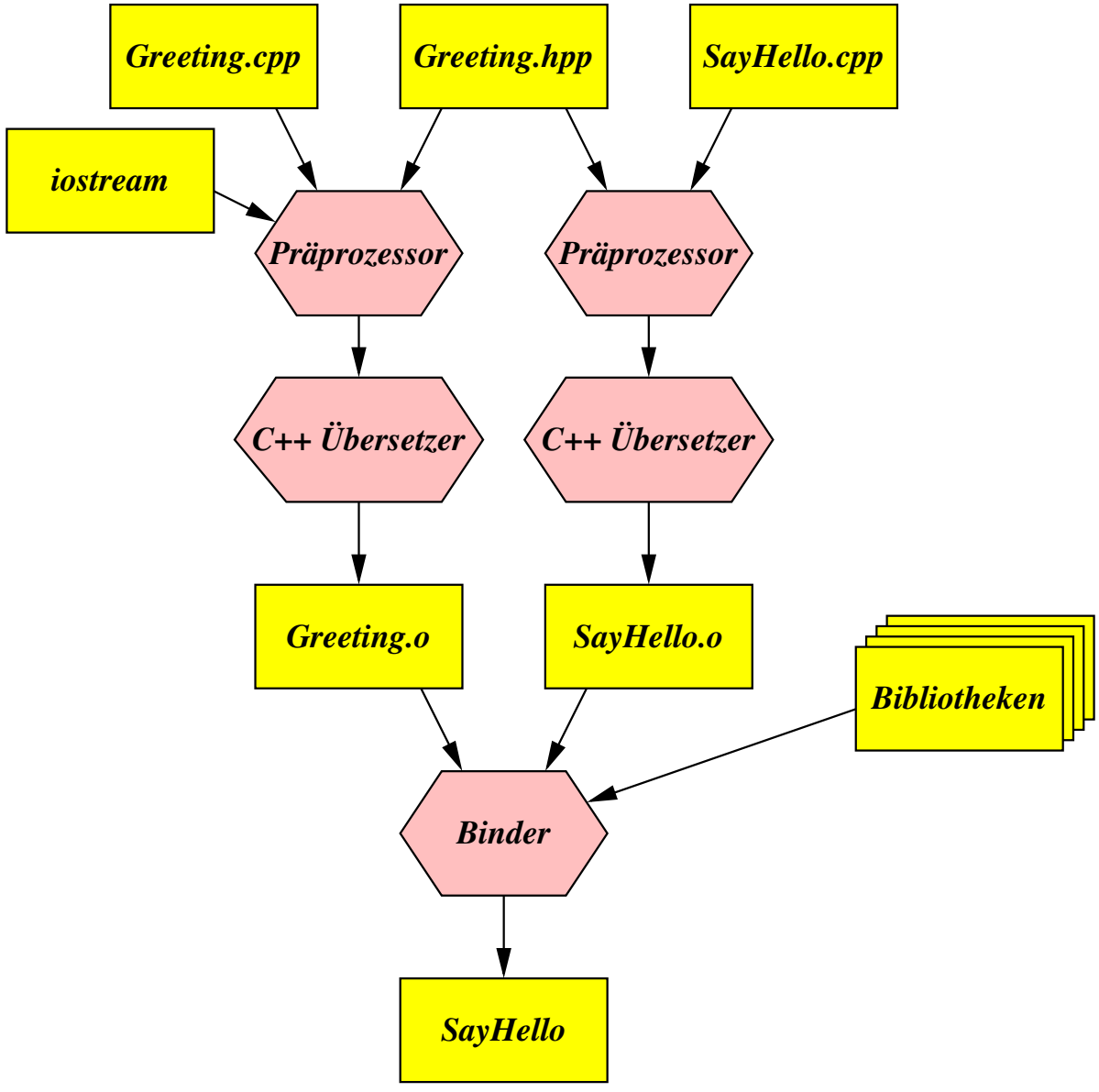
- Dank dem Erbe von C müssen nicht alle Funktionen einer Klasse zugeordnet werden.
- In der Tat darf die Funktion *main*, bei der die Ausführung (nach der Konstruktion globaler Variablen) startet und die Bestandteil eines jeden Programmes sein muss, nicht innerhalb einer Klasse definiert werden.
- Sobald *main* beendet ist, wird das Ende der gesamten Programmausführung eingeleitet.
- Der ganzzahlige Wert, den *main* zurückgibt, wird der Ausführungsumgebung zurückgegeben. Entsprechend den UNIX-Traditionen steht hier 0 für Erfolg und andere Werte deuten ein

SayHello.cpp

```
int main() {
   Greeting greeting;
   greeting.hello();
   return 0;
} // main()
```

- Mit *Greeting greeting* wird eine lokale Variable mit dem Namen *greeting* und dem Datentyp *Greeting* definiert. Das entsprechende Objekt wird hier automatisch instantiiert, sobald *main* startet.
- Durch *greeting.hello*() wird die Methode *hello* für das Objekt *greeting* aufgerufen. Die Klammern sind auch dann notwendig, wenn keine Parameter vorkommen.

Bau eines Programms



- Die gängigen Implementierungen für C++ stellen nur eine schwache Form der Schnittstellensicherheit her.
- Diese wird typischerweise erreicht durch das Generieren von Namen, bei denen teilweise die Typinformation mit integriert ist, so dass Objekte gleichen Namens, jedoch mit unterschiedlichen Typen nicht so ohne weiteres zusammengebaut werden.

```
thales$ ls
Greeting.cpp  Greeting.hpp  SayHello.cpp
thales$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ws14/cpp/Makefile
thales$ make depend
gcc-makedepend  Greeting.cpp SayHello.cpp
thales$ make
g++ -Wall -g -std=gnu++11   -c -o Greeting.o Greeting.cpp
g++ -Wall -g -std=gnu++11   -c -o SayHello.o SayHello.cpp
g++ -o SayHello  Greeting.o SayHello.o
thales$ ./SayHello
Hello, fans of C++!
Hello, fans of C++!
thales$ make realclean
rm -f Greeting.o SayHello.o
rm -f SayHello
thales$ ls
Greeting.cpp  Greeting.hpp  Makefile  SayHello.cpp
thales$
```

- *make* ist ein Werkzeug, das eine Datei namens *Makefile* (oder *makefile*) im aktuellen Verzeichnis erwartet, in der Methoden zur Generierung bzw. Regenerierung von Dateien beschrieben werden und die zugehörigen Abhängigkeiten.
- *make* ist dann in der Lage festzustellen, welche Zieldateien fehlen bzw. nicht mehr aktuell sind, um diese dann mit den spezifizierten Kommandos neu zu erzeugen.
- *make* wurde von Stuart Feldman 1979 für das Betriebssystem UNIX entwickelt. 2003 wurde er hierfür von der ACM mit dem Software System Award ausgezeichnet.

```
thales$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ws14/cpp/Makefile
```

- Unter der genannten URL steht eine Vorlage für ein für C++ geeignetes *Makefile* zur Verfügung.
- Das Kommando *wget* lädt Inhalte von einer gegebenen URL in das lokale Verzeichnis.

```
thales$ make depend
```

- Das heruntergeladene *Makefile* geht davon aus, dass Sie den g++ verwenden (GNU C++ Compiler) und die regulären C++-Quellen in ».cpp« enden und die Header-Dateien in ».hpp«.
- Mit dem Aufruf von »make depend« werden die Abhängigkeiten neu bestimmt und im *Makefile* eingetragen. Dies muss zu Beginn mindestens einmal aufgerufen werden.
- Wenn Sie dies nicht auf unseren Rechnern probieren, sollten Sie das hier implizit verwendete Skript *gcc-makedepend* von uns klauen. Es ist in Perl geschrieben und sollte mit jeder üblichen Perl-Installation zurechtkommen. Eine Manualseite steht zur Verfügung.

Vollständige Implementierung im Header

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

#include <iostream>

class Greeting {
   public:
      void hello() {
         std::cout << "Hello, fans of C++!" << std::endl;
      }
      void hi() {
         std::cout << "Hi!" << std::endl;
      }
}; // class Greeting

#endif
```

- Es ist auch möglich, die Methoden innerhalb des Headers direkt zu implementieren.
- Das verlangsamt die Übersetzungszeiten und es ist nicht sichergestellt, dass der Code für die Methodenimplementierungen nur einmal existiert, wenn diese mehrfach per **#include** einkopiert und somit übersetzt werden.

Greeting.hpp

```
inline void hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
}
```

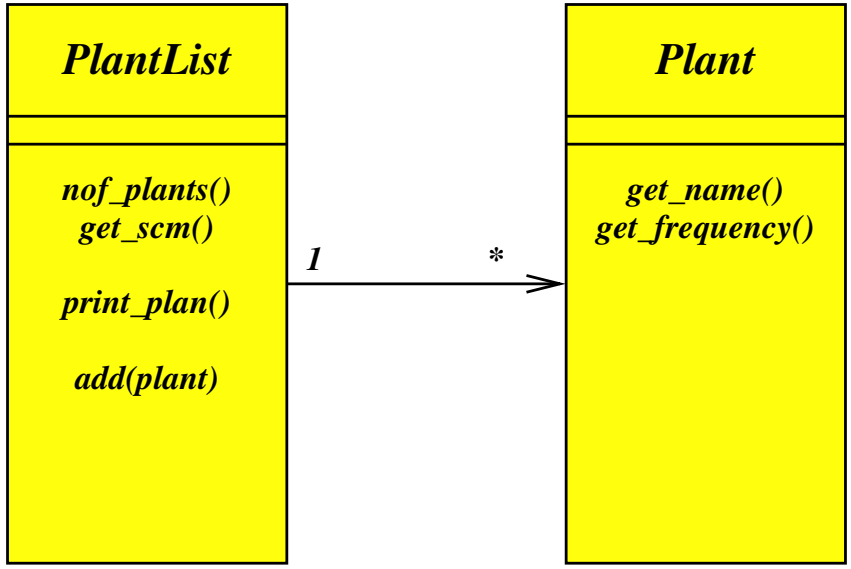
- Es ist auch möglich, dem Übersetzer nahezulegen, auf den Methodenaufruf zu verzichten und stattdessen diesen mit der Implementierung der Methode zu ersetzen.
- Ob dies sinnvoll ist, hängt u.a. auch davon ab, wie umfangreich die Methode ist.
- Das ist nicht möglich mit Methoden, deren zugehörige Implementierung zur Laufzeit gesucht wird (dynamischer Polymorphismus).

Lebensort und -dauer eines Objekts

SayHello.cpp

```
#include "Greeting.hpp"

Greeting greeting1;

int main() {
   greeting1.hello();

   Greeting greeting2;
   greeting2.hello();

   Greeting* greeting3 = new Greeting();
   greeting3->hello();
   delete greeting3;
} // main()
```

- Global erzeugte Objekte wie *greeting1* werden vor dem Aufruf von *main* erzeugt und erst nach dem Verlassen von *main* abgebaut.
- Lokale Variablen wie *greeting2* werden jedesmal erzeugt, wenn der umgebende Block erzeugt wird und beim Verlassen des Blocks automatisch abgebaut.
- Mit **new** kann ein Objekt dynamisch auf dem Heap erzeugt werden. Dieses existiert, bis es explizit mit **delete** wieder abgebaut wird.



- Die Aufgabenstellung ist die Generierung eines tageweisen Bewässerungsplans für eine Menge von Pflanzen, von denen jede unterschiedliche Bewässerungsfrequenzen bevorzugt.
- *scm* steht für das *kleinste gemeinsame Vielfache* (engl. *smallest common multiple*; kurz: kgV) und *get_scm* liefert das kgV aller erfasster Bewässerungsfrequenzen zurück. Nach dieser Zahl von Tagen wiederholt sich der Bewässerungsplan.

Plant.hpp

```
#ifndef PLANT_H
#define PLANT_H

#include <string>

class Plant {
   public:
      // constructors
      Plant(std::string plantName, int wateringFrequency);
         // PRE: wateringFrequency >= 1
      Plant(const Plant &plant);

      // accessors
      std::string get_name() const;
      int get_frequency() const;

   private:
      std::string name;
      int frequency;
};

#endif
```

Plant.cpp

```
#include <cassert>
#include "Plant.hpp"

Plant::Plant(std::string plantName, int wateringFrequency) :
   name(plantName),
   frequency(wateringFrequency) {
   assert(wateringFrequency >= 1);
} // Plant::Plant

Plant::Plant(const Plant &plant) :
   name(plant.name),
   frequency(plant.frequency) {
} // Plant::Plant

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

int Plant::get_frequency() const {
   return frequency;
} // Plant::get_frequency
```

Plant.hpp

```
class Plant {
   public:
      // constructors
      Plant(const std::string& plantName, int wateringFrequency);
         // PRE: wateringFrequency >= 1
      // ...

   private:
      const std::string name;
      int frequency;
};
```

Plant.cpp

```
Plant::Plant(const string& plantName, int wateringFrequency) :
      name{plantName}, frequency{wateringFrequency} {
   assert(wateringFrequency >= 1);
}
```

- Wenn sich Variablen wie hier *name* nie verändern, dann ist es sinnvoll, sie mit **const** zu vereinbaren. Solche Variablen müssen dann zwingend über einen ⟨ctor-initializer⟩ initialisiert werden.

PlantList.hpp

```
#ifndef PLANTLIST_H
#define PLANTLIST_H

#include <list>
#include "Plant.hpp"

class PlantList {
   public:
      // constructors
      PlantList();
      // accessors
      int nof_plants() const;
      int get_scm() const; // PRE: nof_plants() > 0
      // printing
      void print_plan(int day); // PRE: day >= 0
      void print_plan();
      // mutators
      void add(Plant plant);
   private:
      std::list< Plant > plants;
      int scm; // of watering frequencies
};
#endif
```

PlantList.hpp

```
#include <list>
#include "Plant.hpp"

// ...

std::list<Plant> plants;
```

- Dies deklariert *plants* als eine Liste von Elementen des Typs *Plant*.
- *std*::*list* ist eine Template-Klasse, bei der der Elementtyp als Typparameter übergeben wird.
- Entsprechend wird hier nicht nur *plants* deklariert, sondern auch implizit eine neue Klasse ausgehend von *std*::*list* erzeugt. Dies wird auch als Instantiierung eines Templates bezeichnet (*template instantiation*).
- Diese Template-Klasse gehört zur STL (*standard template library*), die Bestandteil von ISO C++ ist.

PlantList.cpp

```
void PlantList::print_plan(int day) const {
   assert(day >= 0);
   for (auto& plant: plants) {
      if (day % plant.get_frequency() == 0) {
         cout << plant.get_name() << endl;
      }
   }
}
```

- Seit C++11 gibt es eine spezielle Form der **for**-Schleife, die die Iteration einer Datenstruktur auf elegantem Wege erlaubt.
- Durch die Verwendung von **auto** wird dem Übersetzer die Herleitung des entsprechenden Typs überlassen.
- Durch die Verwendung von „&“ wird hier mit Referenzen gearbeitet und nicht mit Kopien.
- In jedem Schleifendurchlauf ist somit *plant* eine Referenz auf das aktuelle Element aus der Liste *plants*.

```
for (list<Plant>::const_iterator iterator = plants.cbegin();
      iterator != plants.cend(); ++iterator) {
   if (day % iterator->get_frequency() == 0) {
      cout << iterator->get_name() << endl;
   }
}
```

- Die elegante Form der **for**-Schleife wird intern durch Iteratoren umgesetzt, die ähnlich wie Array-Zeiger verwendet werden können, da all die entsprechenden Operatoren für sie überladen sind.
- *list*<*Plant*>::*iterator* ist eine Klasse, die innerhalb der Klasse *list*<*Plant*> enthalten ist.
- Innerhalb des Initialisierungsteils der **for**-Schleife wird hier *iterator* als Objekt dieser speziellen Iterator-Klasse deklariert und mit *plants.begin*() initialisiert, das einen auf das erste Element zeigenden Iterator zurückliefert.

PlantList.cpp

```
void PlantList::add(Plant plant) {
   int frequency = plant.get_frequency();
   if (scm == 0) {
      scm = frequency;
   } else if (scm % frequency != 0) {
      // computing smallest common multiple using Euclid
      int x0 = scm, x = scm, y0 = frequency, y = frequency;
      while (x != y) {
         if (x > y) {
            y += y0;
         } else {
            x += x0;
         }
      }
      scm = x;
   }
   plants.push_back(plant);
}
```

- *plants.push_back*(*plant*) belegt Speicher für eine Kopie von *plant* und hängt diese Kopie an das Ende der Liste ein.

Hauptprogramm

WateringPlan.cpp

```
#include <iostream>
#include <string>
#include "Plant.hpp"
#include "PlantList.hpp"

int main() {
   PlantList plants;
   std::string name; int frequency;

   while (std::cin >> name >> frequency) {
      plants.add(Plant{name, frequency});
   }
   plants.print_plan();
}
```

WateringPlan.cpp

```
while (std::cin >> name >> frequency) {
   plants.add(Plant{name, frequency});
}
```

- Normalerweise liefert *cin* >> *name* den Wert von *cin* zurück, um eine Verkettung von Eingabe-Operationen für den gleichen Eingabestrom zu ermöglichen.
- Hier jedoch findet implizit eine Konvertierung statt, da ein **bool**-Wert benötigt wird. Dies gelingt u.a. mit Hilfe eines sogenannten Konvertierungs-Operators der entsprechenden Klasse.
- Entsprechend ist die Bedingung genau dann wahr, falls beide Lese-Operationen erfolgreich sind.
- *Plant*{*name*, *frequency*} erzeugt ein sogenanntes temporäres Objekt des Typs *Plant*, das vollautomatisch wieder aufgeräumt wird, sobald die Auswertung des entsprechenden Ausdrucks abgeschlossen ist.

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*&.

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

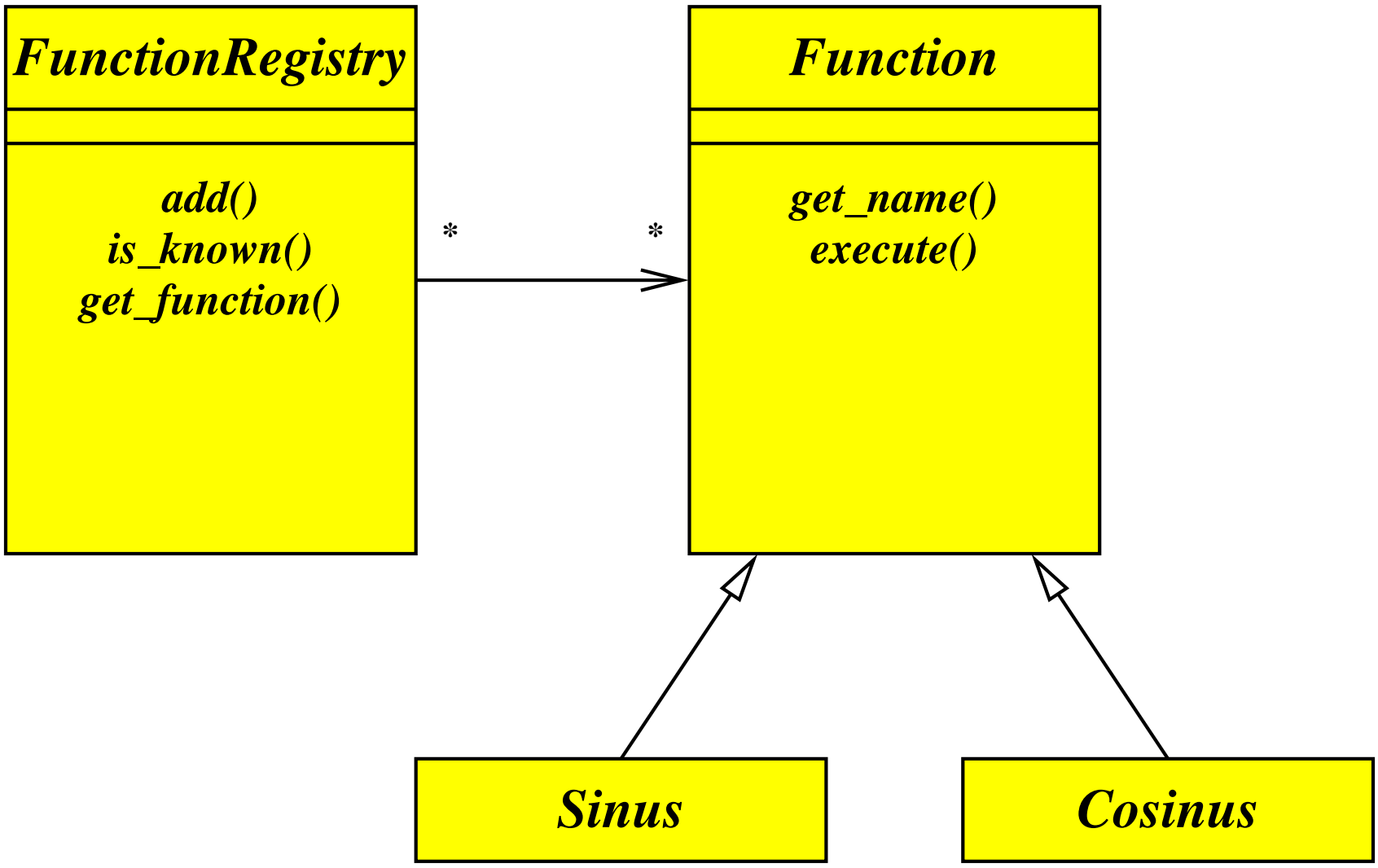
```
#include <iostream>
#include "Sinus.hpp"

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
```