Übungen zu Parallele Programmierung mit C++
Einführung in C++
SS 2017

Andreas F. Borchert

Universität Ulm

12. Mai 2017

- 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⟩

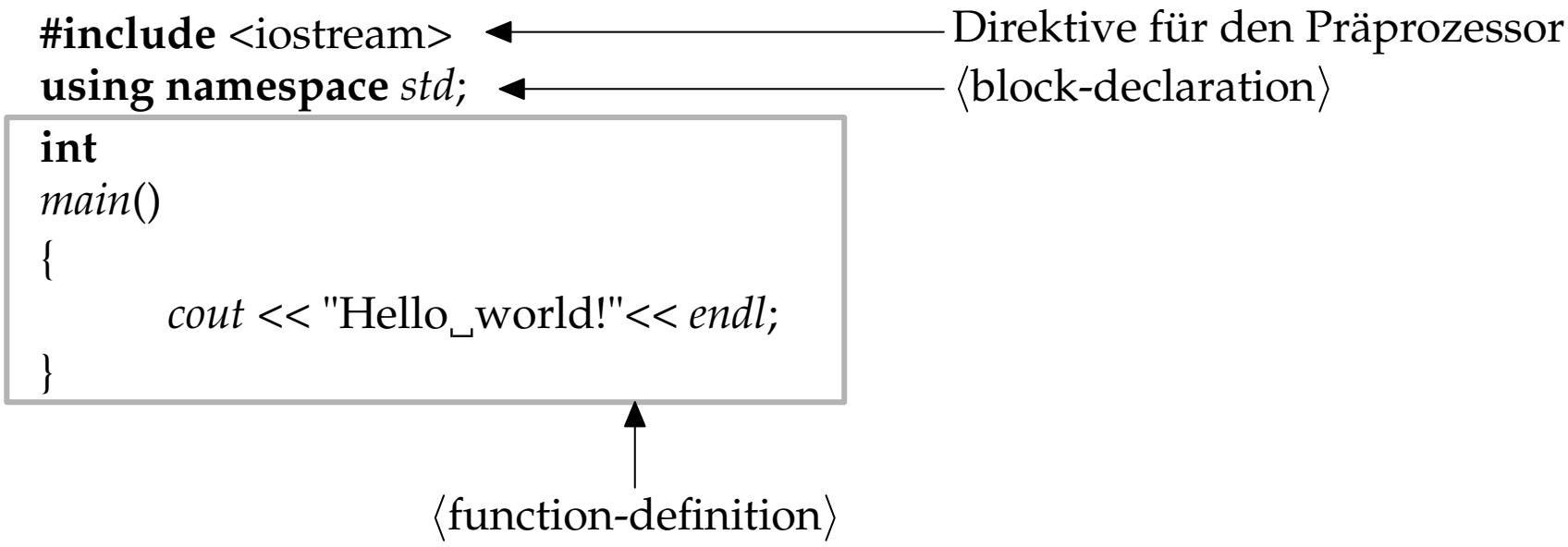
⟶ ⟨namespace-definition⟩

⟶ ⟨empty-declaration⟩

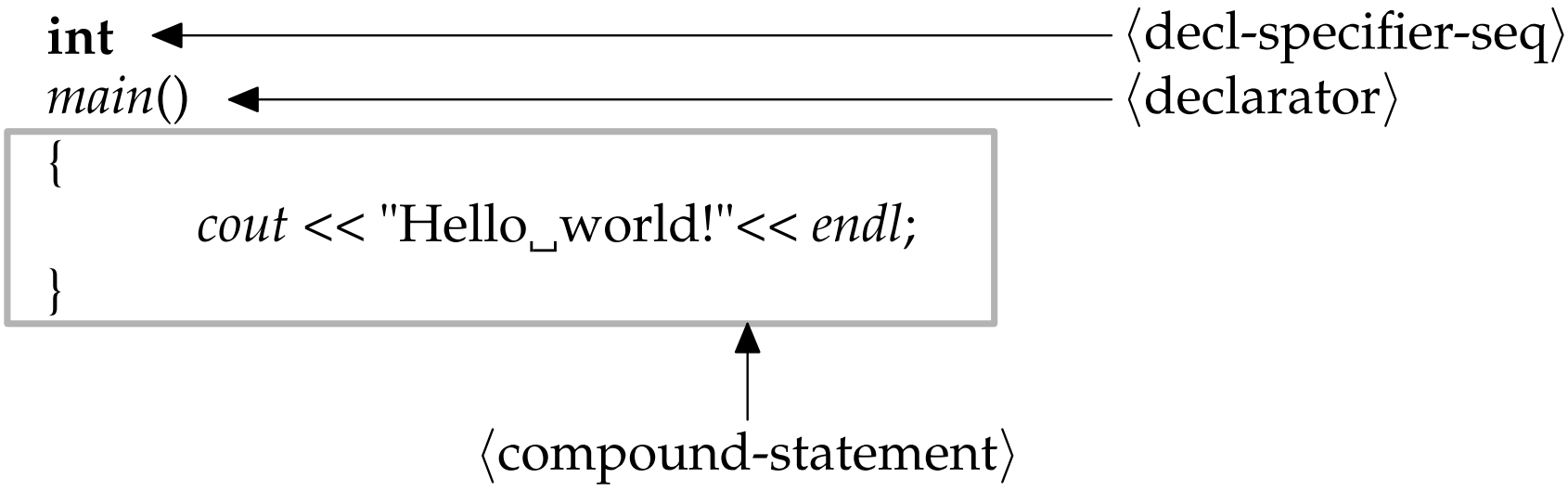
⟶ ⟨attribute-declaration⟩

- 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 Operanden einen *ostream* und als rechten Operanden 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()
```

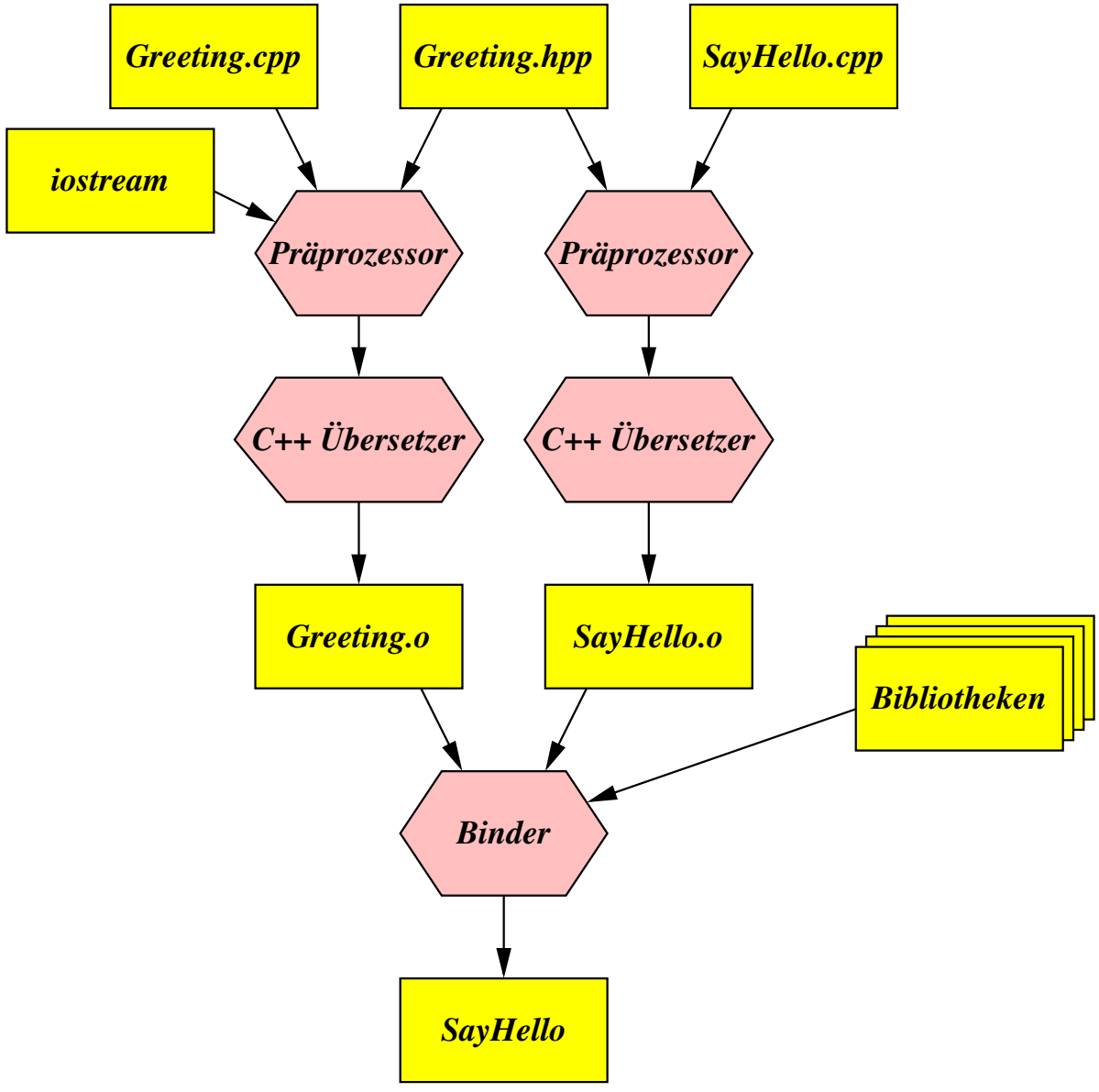
- Dank dem Erbe von C müssen nicht alle Funktionen einer Klasse zugeordnet werden.
- In der Tat darf die Funktion *main*, bei der die Ausführung (nach der Konstruktion globaler Variablen) startet und die Bestandteil eines jeden Programmes sein muss, nicht innerhalb einer Klasse definiert werden.
- Sobald *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.

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.

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

- Seit C++11 wird die {...}-Notation (⟨braced-init-list⟩) 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$
```

Übergabe von Objekten per Referenz


```
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.

Klassen mit Zeigern

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** uninitialisiert 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.)

Verschiebekonstruktor für das Array

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.

Zugriff auf ein Array-Element

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;
```

array.hpp

```
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*&.

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
```

- 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

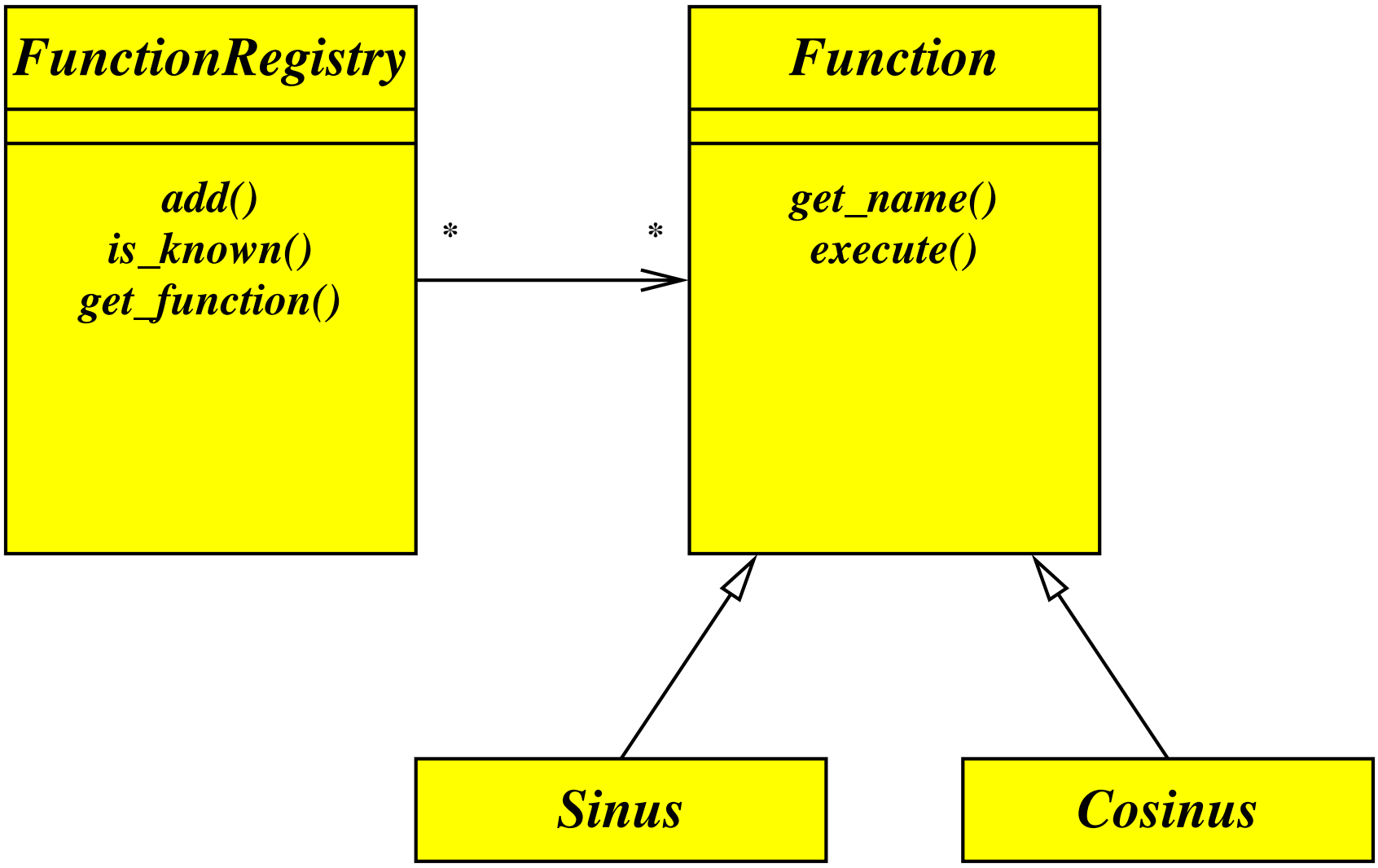
```
#include <iostream>
#include "Sinus.hpp"

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
```

- 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.

- Das Lambda-Kalkül geht auf Alonzo Church und Stephen Kleene zurück, die in den 30er-Jahren damit ein formales System für berechenbare Funktionen entwickelten.
- Zu den wichtigsten Arbeiten aus dieser Zeit gehört der Aufsatz von Alonzo Church: *An Unsolvable Problem of Elementary Number Theory*, *Americal Journal of Mathematics*, Band 58, Nr. 2 (April 1936), S. 345–363.
- Diese Arbeit zeigt, dass es keine berechenbare Funktion gibt, die die Äquivalenz zweier Ausdrücke des Lambda-Kalküls feststellen kann.
- Die Turing-Maschine und das Lambda-Kalkül sind in Bezug auf die Berechenbarkeit äquivalent.
- Das Lambda-Kalkül wurde von funktionalen Programmiersprachen übernommen (etwa von Lisp und Scheme) und wird auch gerne zur formalen Beschreibung der Semantik einer Programmiersprache verwendet (denotationelle Semantik).

Es gibt zwei wesentliche Punkte, weswegen Lambda-Ausdrücke auch in nicht-funktionalen Programmiersprachen (wie etwa C++) interessant sind:

- Anonyme Funktionen können lokal konstruiert und übergeben werden. Ein Beispiel dafür wäre das Sortierkriterium bei *sort*. Die lokal definierte anonyme Funktion kann dabei auch die Variablen der sie umgebenden Funktion sehen (*closure*).
- Funktionen können aus anderen Funktionen abgeleitet werden. Beispielsweise kann eine Funktion mit zwei Argumenten in eine Funktion abgebildet werden, bei der der eine Parameter fest vorgegeben und nur noch der andere variabel ist (*currying*).

Grundsätzlich kann das alles auch konventionell formuliert werden durch explizite Klassendefinitionen. Aber dann wird der Code umfangreicher, umständlicher (etwa durch die explizite Übergabe der lokal sichtbaren Variablen) und schwerer lesbarer (zusammenhängender Code wird auseinandergerissen). Allerdings können Lambda-Ausdrücke auch zur Unlesbarkeit beitragen.

Da in C++ Funktionsobjekte wegen der entsprechenden STL-Algorithmen recht beliebt sind, gab es mehrere Ansätze, Lambda-Ausdrücke in C++ einzuführen:

- *boost*::*lambda* von Jaakko Järvi, entwickelt von 1999 bis 2004
- *boost*::*phoenix* von Joel de Guzman und Dan Marsden, entwickelt von 2002 bis 2005
- Integration von Lambda-Ausdrücken in den C++-Standard ISO-14882-2012.

transform2.cpp

```
int main() {
   list<int> ints;
   for (int i = 1; i <= 10; ++i) {
      ints.push_back(i);
   }

   list<int> squares;
   transform(ints.begin(), ints.end(),
      back_inserter(squares), [](int val) { return val*val; });

   for (int val: squares) {
      cout << val << endl;
   }
}
```

- Mit Lambda-Ausdrücken werden implizit unbenannte Klassen erzeugt und temporäre Objekte instantiiert.
- In diesem Beispiel ist [](**int** *val*){ **return** *val*∗*val*; } der Lambda-Ausdruck, der ein temporäres unäres Funktionsobjekt erzeugt, das sein Argument quadriert.

⟨lambda-expression⟩ ⟶ ⟨lambda-introducer⟩ [⟨lambda-declarator⟩]
⟨compound-statement⟩

⟨lambda-introducer⟩ ⟶ „[“ [⟨lambda-capture⟩] „]“

⟨lambda-capture⟩ ⟶ ⟨capture-default⟩

⟶ ⟨capture-list⟩

⟶ ⟨capture-default⟩ „,“ ⟨capture-list⟩

⟨capture-default⟩ ⟶ „&“ | „=“

⟨capture-list⟩ ⟶ ⟨capture⟩ [„...“]

⟶ ⟨capture-list⟩ „,“ ⟨capture⟩ [„...“]

⟨capture⟩ ⟶ ⟨simple-capture⟩
⟶ ⟨init-capture⟩
⟨simple-capture⟩ ⟶ ⟨identifier⟩
⟶ „&“ ⟨identifier⟩
⟶ **this**
⟨init-capture⟩ ⟶ ⟨identifier⟩ ⟨initializer⟩
⟶ „&“ ⟨identifier⟩ ⟨initializer⟩
⟨lambda-declarator⟩ ⟶ „(“ ⟨parameter-declaration-clause⟩ „)“
[**mutable**] [⟨exception-specification⟩]
[⟨attribute-specifier-seq⟩]
[⟨trailing-return-type⟩]

- Die ⟨init-capture⟩ kommt mit dem C++14-Standard hinzu.

- Lambda-Ausdrücke, die in eine Funktion eingebettet sind, „sehen“ die lokalen Variablen aus den umgebenden Blöcken.
- In vielen funktionalen Programmiersprachen überleben die lokalen Variablen selbst dann, wenn der sie umgebende Block verlassen wird, weil es noch überlebende Funktionsobjekte gibt, die darauf verweisen. Dies benötigt zur Implementierung sogenannte *cactus stacks*.
- Da für C++ der Aufwand für diese Implementierung zu hoch ist und auch die Übersetzung normalen Programmtexts ohne Lamda-Ausdrücke verteuern würde, fiel die Entscheidung, einen alternativen Mechanismus zu entwickeln, der über *lambda-capture* spezifiziert wird.

Kopien der Variablen aus der Umgebung


```
template<typename T>
function<T(T)> create_multiplier(T factor) {
   return function<T(T)>([=](T val) { return factor*val; });
}

int main() {
   auto multiplier = create_multiplier(7);
   for (int i = 1; i < 10; ++i) {
      cout << multiplier(i) << endl;
   }
}
```

- *create_multiplier* ist eine Template-Funktion, die ein mit einem vorgegebenen Faktor multiplizierendes Funktionsobjekt erzeugt und zurückliefert.
- Die Standard-Template-Klasse *function* wird hier genutzt, um das Funktionsobjekt in einen bekannten Typ zu verpacken.
- Die *lambda-capture* [=] legt fest, dass die aus der Umgebung referenzierten Variablen beim Erzeugen des Funktionsobjekts kopiert werden.

Erzeugung einer unbenannten Klasse 83

```
template<typename T>
class Anonymous {
   public:
      Anonymous(const T& factor_) : factor(factor_) {}
      T operator()(T val) const {
         return factor*val;
      }
   private:
      T factor;
};

template<typename T>
function<T(T)> create_multiplier(T factor) {
   return function<T(T)>(Anonymous<T>(factor));
}
```

- Der Lambda-Ausdruck führt implizit zu einer Erzeugung einer unbenannten Klasse (hier einfach *Anonymous* genannt).
- Jeder aus der Umgebung referenzierte Variable, die kopiert wird, findet sich als gleichnamige Variable der Klasse wieder, die bei der Konstruktion übergeben wird.

```
template<typename T>
tuple<function<T()>, function<T()>, function<T()>>
create_counter(T val) {
   shared_ptr<T> p(new T(val));
   auto incr = [=]() { return ++*p; };
   auto decr = [=]() { return --*p; };
   auto getval = [=]() { return *p; };
   return make_tuple(function<T()>(incr),
      function<T()>(decr), function<T()>(getval));
}
```

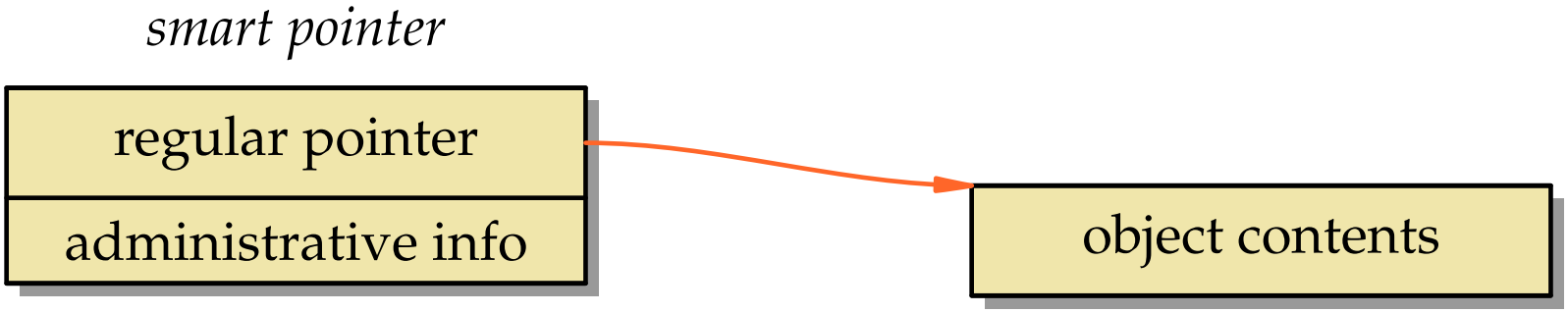
- In funktionsorientierten Sprachen werden gerne die gemeinsamen Variablen aus der Hülle benutzt, um private Variablen für eine Reihe von Funktionsobjekten zu haben, die wie objekt-orientierte Methoden arbeiten.
- Das ist auch in C++ möglich mit Hilfe von *shared_ptr*.
- Aber normalerweise ist es einfacher, eine entsprechende Klasse zu schreiben.

```
int main() {
   function<int()> incr, decr, getval;
   tie(incr, decr, getval) = create_counter(0);
   char ch;
   while (cin >> ch) {
      switch (ch) {
         case '+': incr(); break;
         case '-': decr(); break;
         default: break;
      }
   }
   cout << getval() << endl;
}
```

- *create_counter* erzeugt ein Tupel (Datenstruktur aus **#include** <tuple>) und *tie* erlaubt es, gleich mehrere Variablen aus einem Tupel zuzuweisen.
- Danach bleibt die gemeinsame private Variable solange bestehen, bis diese von *shared_ptr* freigegeben wird, d.h. sobald die letzte Referenz darauf verschwindet.

```
vector<int> values(10);
int count = 0;
generate(values.begin(), values.end(), [&]() { return ++count; });
```

- Alternativ können Variablen nicht kopiert, sondern per impliziter Referenz benutzt werden.
- Dann darf das Funktionsobjekt aber nicht länger leben bzw. benutzt werden, als die entsprechenden Variablen noch leben. Das liegt in der Verantwortung des Programmierers.
- *generate* steht über **#include** <algorithm> zur Verfügung und weist die von dem Funktionsobjekt erzeugten Werte sukzessiv allen referenzierten Werten zwischen dem ersten Iterator (inklusive) und dem zweiten Iterator (exklusive) zu.



- Intelligente Zeiger (*smart pointers*) entsprechen weitgehend normalen Zeigern, haben aber Sonderfunktionalitäten aufgrund weiterer Verwaltungsinformationen.
- Sie werden insbesondere dort eingesetzt, wo die Sprache selbst keine Infrastruktur für die automatisierte Speicherfreigabe anbietet.
- Sie unterstützen das RAII-Prinzip für Zeiger.

- Seit dem C++11-Standard sind intelligente Zeiger Bestandteil der C++-Bibliothek. Zuvor gab es nur den inzwischen abgelösten *auto_ptr* und die Erweiterungen der Boost-Library, die jetzt praktisch übernommen worden sind.
- C++11 bietet folgende Varianten an:

unique_ptr	nur ein Zeiger auf ein Objekt
shared_ptr	mehrere Zeiger auf ein gemeinsames Objekt mit externem Referenzzähler
weak_ptr	nicht das Überleben sichernder „schwacher“ Zeiger auf ein Objekt mit externem Referenzzähler

- Grundsätzlich sollte ein mit **new** erzeugtes Objekt mit **delete** wieder freigegeben werden, sobald der letzte Verweis entfernt wird.
- Unterbleibt dies, haben wir ein Speicherleck.
- Wichtig ist aber auch, dass kein Objekt mehrfach freigegeben wird. Dies kann bei manueller Freigabe leicht geschehen, wenn es mehrere Zeiger auf ein Objekt gibt.
- Intelligente Zeiger können sich auch dann um eine korrekte Freigabe kümmern, wenn eine Ausnahmenbehandlung ausgelöst wird.
- Jedoch können zyklische Datenstrukturen mit der Verwendung von Referenzzählern alleine nicht korrekt aufgelöst werden. Hier sind ggf. Ansätze mit sogenannten „schwachen“ Zeigern denkbar.

Grundregeln im Umgang mit intelligenten Zeigern 90

- Auf ein Objekt sollten nur Zeiger eines Typs verwendet werden.
- Die einzige Ausnahme davon ist die Mischung von *shared_ptr* und *weak_ptr*.
- Im Normalfall bedeutet dies, dass die entsprechenden Klassen angepasst werden müssen, da es dann nicht mehr zulässig ist, **this** zurückzugeben.
- Üblicherweise sollte sogleich bei dem Entwurf einer Klasse geplant werden, welche Art von Zeigern zum Einsatz kommt.
- Normalerweise sollten entsprechend des RAII-Prinzips „nackte“ Zeiger außerhalb isolierter Fälle konsequent vermieden werden.

- Wenn es nur einen einzigen Zeiger auf ein Objekt geben soll, dann empfiehlt sich die Verwendung von *unique_ptr*.
- Das ist besonders geeignet für lokale Zeigervariablen oder Zeiger innerhalb einer Klasse.
- Die Freigabe erfolgt dann vollautomatisch, sobald der zugehörige Block bzw. das umgebende Objekt freigegeben werden.
- Bei einer Zuweisung wird der Besitz des Zeigers übertragen. Das funktioniert nur entsprechend mit einem sogenannten *move assignment*, d.h. der Zeigerwert wird von einem anderen *unique_ptr*-Objekt gerettet, der im nächsten Moment ohnehin dekonstruiert wird.
- Andere Zuweisungen dieser Zeiger sind nicht möglich, da dies die Restriktion des exklusiven Zugangs verletzen würde.

ptrex.cpp

```
void f(int i) {
   Object* ptr = new Object(i);
   if (i == 2) {
      throw something();
   }
   delete ptr;
}
```

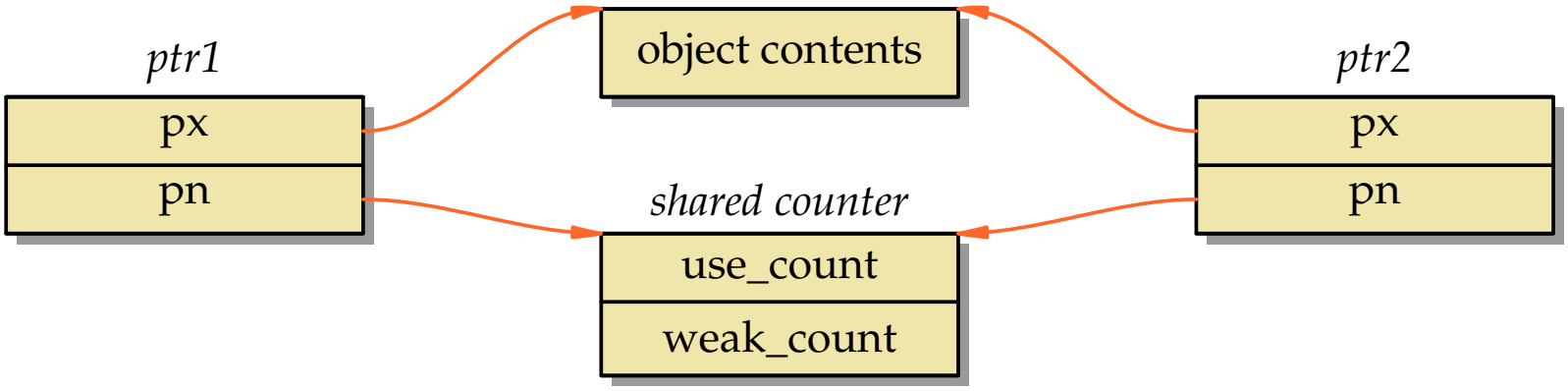
- Wenn Objekte in einer Funktion nur lokal erzeugt und verwendet werden, ist darauf zu achten, dass die Freigabe nicht vergessen wird.
- Dies passiert jedoch leicht bei Ausnahmenbehandlungen (möglicherweise durch eine aufgerufene Funktion) oder bei frühzeitigen **return**-Anweisungen.

ptrex2.cpp

```
void f(int i) {
   unique_ptr<Object> ptr(new Object(i));
   if (i == 2) {
      throw something();
   }
}
```

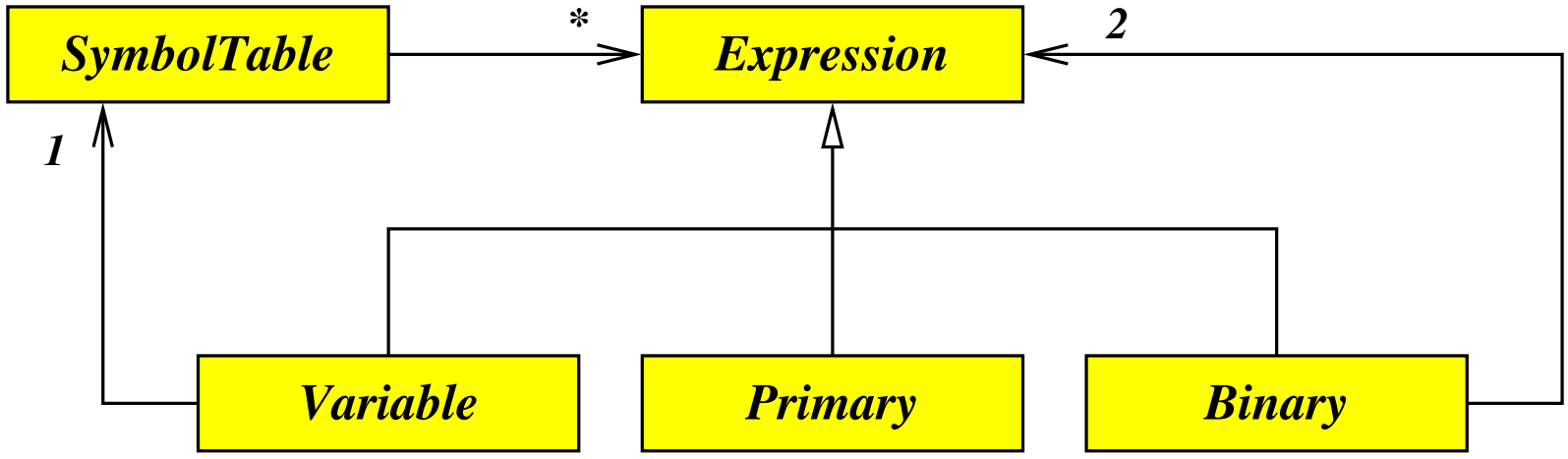
- *ptr* kann hier wie ein normaler Zeiger verwendet werden, abgesehen davon, dass eine Zuweisung an einen anderen Zeiger nicht zulässig ist.
- Dann erfolgt die Freigabe des Objekts vollautomatisch über den Dekonstruktor.

Mehrere Zeiger auf ein Objekt



- Für den allgemeinen Einsatz empfiehlt sich die Verwendung von *shared_ptr*, das mit Referenzzählern arbeitet.
- Zu jedem referenzierten Objekt gehört ein intern verwaltetes Zählerobjekt, das die Zahl der Verweise zählt. Sobald *use_count* auf 0 sinkt, erfolgt die Freigabe des Objekts.
- Das Zählerobjekt wird erst freigegeben, wenn neben *use_count* auch *weak_count* auf 0 sinkt.
- Es ist darauf zu achten, dass für jedes Objekt nur ein gemeinsames Zählerobjekt existiert.

Beispiel: Syntaxbäume



- Für einen Taschenrechner haben wir eine Datenstruktur für Syntaxbäume (*Expression*) mit den abgeleiteten Klassen *Variable*, *Primary* und *Binary*.
- Neben einem Zeiger auf den gerade eingelesenen Ausdrucksbaum kommt hier eine Symboltabelle hinzu, deren Einträge ebenfalls auf Syntaxbäume verweisen. In Abhängigkeit der Benutzereingaben können einzelne Syntaxbäume so vielfach in der Datenstruktur eingebettet sein.
- Entsprechend gibt es keinen Überblick der Verweise auf einen Syntaxbaum und somit dürfen diese erst freigegeben werden, wenn der letzte Verweis wegfällt.

expression.hpp

```
class Expression {
   public:
      virtual ~Expression() {};
      virtual Value evaluate() const = 0;
};

using ExpressionPtr = std::shared_ptr<Expression>;
```

- Bei Klassenhierarchien, bei denen polymorphe Zeiger eingesetzt werden, ist die Deklaration eines virtuellen Dekonstruktors essentiell.
- Die Methode *evaluate* soll den durch den Baum repräsentierten Ausdruck rekursiv auswerten.
- *ExpressionPtr* wird hier als intelligenter Zeiger auf *Expression* definiert, bei dem beliebig viele Zeiger des gleichen Typs auf ein Objekt verweisen dürfen.

expression.hpp

```
class Binary: public Expression {
   public:
      typedef Value (*BinaryOp)(Value val1, Value val2);
      Binary(BinaryOp op, ExpressionPtr expr1,
            ExpressionPtr expr2) :
               op{op}, expr1{expr1}, expr2{expr2} {
      }
      virtual Value evaluate() const {
         return op(expr1->evaluate(), expr2->evaluate());
      }
   private:
      BinaryOp op;
      ExpressionPtr expr1;
      ExpressionPtr expr2;
};
```

- *Binary* repräsentiert einen Knoten des Syntaxbaums mit einem binären Operator und zwei Operanden.
- Statt *Expression*∗ wird dann konsequent *ExpressionPtr* verwendet.

expression.hpp

```
class Variable: public Expression {
   public:
      Variable(SymbolTable& symtab, const std::string& varname) :
         symtab{symtab}, varname{varname} {
      }
      virtual Value evaluate() const {
         ExpressionPtr expr = symtab.get(varname);
         return expr? expr->evaluate(): Value();
      }
      void set(ExpressionPtr expr) {
         symtab.define(varname, expr);
      }
   private: SymbolTable& symtab; std::string varname;
};
using VariablePtr = std::shared_ptr<Variable>;
```

- Die Kompatibilität innerhalb der *Expression*-Hierarchie überträgt sich auch auf die zugehörigen intelligenten Zeiger. Zwar bilden die intelligenten Zeigertypen keine formale Hierarchie, aber sie bieten Zuweisungs-Operatoren auch für fremde Datentypen an, die nur dann funktionieren, wenn die Kompatibilität für die entsprechenden einfachen Zeigertypen existiert.

parser.cpp

```
ExpressionPtr Parser::parseSimpleExpression() throw(Exception) {
   ExpressionPtr expr = parseTerm();
   while (getToken().symbol == Token::PLUS ||
          getToken().symbol == Token::MINUS) {
      Binary::BinaryOp op;
      switch (getToken().symbol) {
         case Token::PLUS: op = addop; break;
         case Token::MINUS: op = subop; break;
         default: /* does not happen */ break;
      }
      nextToken();
      ExpressionPtr expr2 = parseTerm();
      expr = std::make_shared<Binary>(op, expr, expr2);
   }
   return expr;
}
```

- *make_shared* erzeugt ein Objekt des angegebenen Typs mit **new** und liefert den passenden intelligenten Zeigertyp zurück.
- Das ist in diesem Beispiel *shared_ptr<Binary>*, das entsprechend der Klassenhierarchie an den allgemeinen Zeigertyp *ExpressionPtr* zugewiesen werden kann.

Typentests und Konvertierungen bei intelligenten Zeigern

parser.cpp

```
ExpressionPtr Parser::parseAssignment() throw(Exception) {
   ExpressionPtr expr = parseSimpleExpression();
   if (getToken().symbol == Token::BECOMES) {
      VariablePtr var = std::dynamic_pointer_cast<Variable>(expr);
      if (!var) {
         throw Exception(getToken(), "variable expected");
      }
      nextToken();
      ExpressionPtr expr2 = parseSimpleExpression();
      var->set(expr2);
      return expr2;
   }
   return expr;
}
```

- Statt **dynamic_cast** ist bei intelligenten Zeigern *dynamic_pointer_cast* zu verwenden, um eine ungewollte Neu-Erzeugung eines Zählerobjekts zu vermeiden.
- Genauso wie bei **dynamic_cast** wird ein Nullzeiger geliefert, falls der angegebene Zeiger nicht den passenden Typ hat.

- Bei Referenzzyklen bleiben die Referenzzähler positiv, selbst wenn der Zyklus insgesamt nicht mehr von außen erreichbar ist.
- Eine automatisierte Speicherfreigabe (*garbage collection*) würde den Zyklus freigeben, aber mit Zeigern auf Basis von *shared_ptr* gelingt dies nicht.
- Eine Lösung für dieses Problem sind sogenannte schwache Zeiger (*weak pointers*), die bei der Referenzzählung nicht berücksichtigt werden.

Beispiel: Doppelt verkettete Liste

list.hpp

```
template <typename T>
class List {
   private:
      struct Element;
      using Link = std::shared_ptr<Element>;
      using WeakLink = std::weak_ptr<Element>;
      struct Element {
         Element(const T& elem);
         T elem;
         Link next;
         WeakLink prev;
      };
      Link head;
      Link tail;
   public:
      class Iterator {
         // ...
      };
      Iterator begin();
      Iterator end();
      void push_back(const T& object);
};
```

Beispiel: Doppelt verkettete Liste

list.hpp

```
typedef std::shared_ptr<Element> Link;
typedef std::weak_ptr<Element> WeakLink;
struct Element {
   Element(const T& elem);
   T elem;
   Link next;
   WeakLink prev;
};
```

- Die einzelnen Glieder einer doppelt verketteten Liste verweisen jeweils auf den Nachfolger und den Vorgänger.
- Wenn mindestens zwei Glieder in einer Liste enthalten ist, ergibt dies eine zyklische Datenstruktur.
- Das kann dadurch gelöst werden, dass für die Rückverweise schwache Zeiger verwendet werden.

Beispiel: Doppelt verkettete Liste

list.tpp

```
template<typename T>
void List<T>::push_back(const T& object) {
   Link ptr = std::make_shared<Element>(object);
   ptr->prev = tail;
   if (head) {
      tail->next = ptr;
   } else {
      head = ptr;
   }
   tail = ptr;
}
```

- Eine Zuweisung von *shared_ptr* an den korrespondierenden *weak_ptr* ist problemlos möglich wie hier bei: $ptr->prev = tail$

Iterator-Klasse mit schwachem Zeiger

list.hpp

```
class Iterator {
   public:
      class Exception: public std::exception {
         public:
            Exception(const std::string& msg);
            virtual ~Exception() noexcept;
            virtual const char* what() const noexcept;
         private:
            std::string msg;
      };
      bool valid();
      T& operator*();
      Iterator& operator++(); // prefix increment
      Iterator operator++(int); // postfix increment
      Iterator& operator--(); // prefix decrement
      Iterator operator--(int); // postfix decrement
      bool operator==(const Iterator& other);
      bool operator!=(const Iterator& other);
   private:
      friend class List;
      Iterator();
      Iterator(WeakLink ptr);
      WeakLink ptr;
};
```

Dereferenzierung eines schwachen Zeigers

list.tpp

```
template<typename T>
T& List<T>::Iterator::operator*() {
   Link p = ptr.lock();
   if (p) {
      return p->elem;
   } else {
      throw Exception("iterator is expired");
   }
}
```

- Ein schwacher Zeiger kann mit Hilfe der *lock*-Methode in einen regulären Zeiger verwandelt werden.
- Wenn das referenzierte Objekt mittlerweile freigegeben wurde, ist der Zeiger 0.

list.tpp

```
template<typename T>
bool List<T>::Iterator::operator==(const Iterator& other) {
   Link p1 = ptr.lock();
   Link p2 = other.ptr.lock();
   return p1 == p2;
}

template<typename T>
bool List<T>::Iterator::operator!=(const Iterator& other) {
   return !(*this == other);
}
```

- Schwache Zeiger können erst dann miteinander verglichen werden, wenn sie zuvor in reguläre Zeiger konvertiert werden.
- Nullzeiger werden hier als äquivalent angesehen.

```
#include <memory>

class Object;
using ObjectPtr = std::shared_ptr<Object>;
class Object: public std::enable_shared_from_this<Object> {
   public:
      ObjectPtr me() {
         return shared_from_this();
      }
};
```

- Die Grundregel, dass auf ein Objekt nur Zeiger eines Typs verwendet werden sollten, stößt auf ein Problem, wenn statt **this** ein passender intelligenter Zeiger zurückzugeben ist.
- Eine Lösung besteht darin, die Klasse von *std*::*enable_shared_from_this* abzuleiten. Dann steht die Methode *shared_from_this* zur Verfügung. Dies wird implementiert, indem im Objekt zusätzlich ein schwacher Zeiger auf das eigene Objekt verwaltet wird.

```
#include <memory>

class Object;
using ObjectPtr = std::shared_ptr<Object>;
class Object {
   public:
      class Key {
         friend class Object;
         Key() {}
      };
      static ObjectPtr create() {
         return std::make_shared<Object>(Key());
      }
      Object(Key&& key) {}
};
```

- Um die Grundregel durchzusetzen, erscheint es gelegentlich sinnvoll, die regulären Konstruktoren zu verbergen.
- **private** dürfen Sie jedoch nicht sein, da *std*::*make_shared* einen passenden öffentlichen Konstruktor benötigt.
- Eine Lösung bietet der *pass key*-Ansatz. Der Konstruktor ist zwar öffentlich, aber ohne privaten Schlüssel nicht benutzbar.