Übungen zu Parallele Programmierung mit C++
Einführung in C++
WS 2013/2014

Andreas F. Borchert

Universität Ulm

16. Oktober 2013

- 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.

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« oder ».h« enden, unterzubringen. Hierbei steht ».h« für Header-Datei bzw. ».hpp« 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.
- 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.

C++ lässt grüssen

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

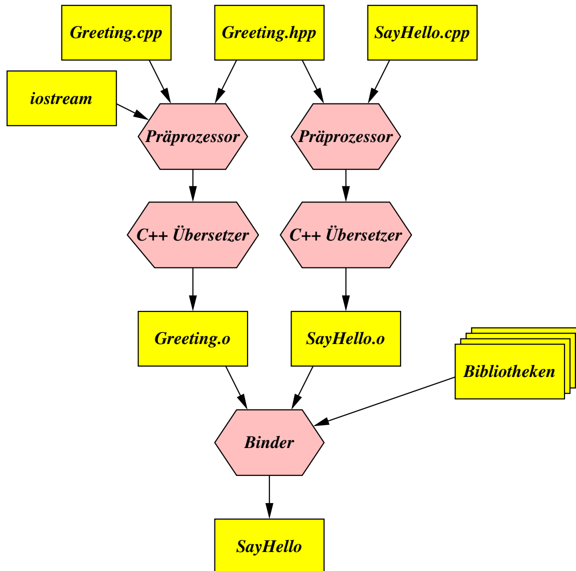
```
#include "Greeting.hpp"

int main() {
   Greeting greeting;
   greeting.hello();
   greeting.hello();
   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 Problem an.

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.


Greeting.cpp
Greeting.hpp
SayHello.cpp
iostream
Präprozessor
Präprozessor
C++ Übersetzer
C++ Übersetzer
Greeting.o
SayHello.o
Bibliotheken
Binder
SayHello

- 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.

Bau eines Programms


```
thales$ ls
Greeting.cpp  Greeting.hpp  SayHello.cpp
thales$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ws12/cpp/cpp/makefile
thales$ sed 's/PleaseRenameMe/SayHello/' <makefile >makefile.tmp &&
> mv makefile.tmp 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$
```

- *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.

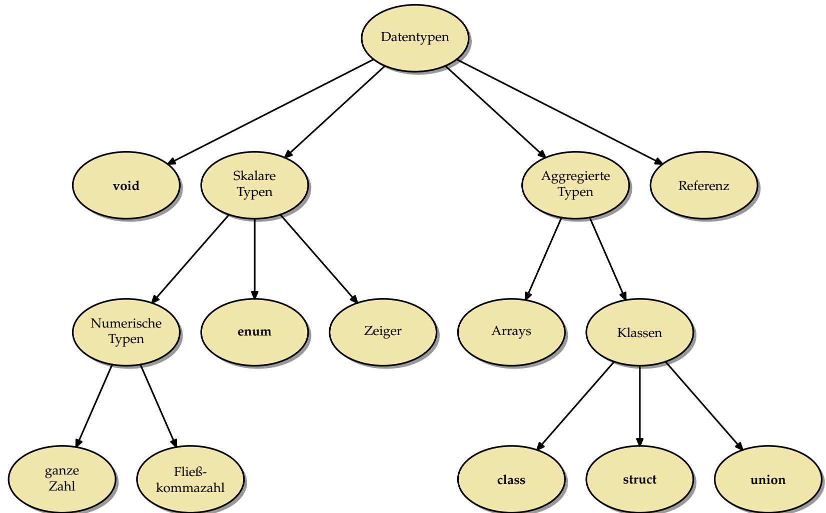
```
thales$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ws13/pp/cpp/makefile
thales$ sed 's/PleaseRenameMe/SayHello/' <makefile >makefile.tmp &&
> mv makefile.tmp makefile
```

- 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.
- In der Vorlage fehlt noch die Angabe, wie Ihr Programm heißen soll. Das wird hier mit dem Kommando *sed* nachgeholt, indem der Text »PleaseRenameMe« entsprechend ersetzt wird.

```
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. Sie finden es auf einem beliebigen unserer Rechner unter »/usr/local/bin/gcc-makedepend«. Es ist in Perl geschrieben und sollte mit jeder üblichen Perl-Installation zurechtkommen.

Typen in C++



- Zu den skalaren Datentypen gehören alle elementaren Typen, die entweder numerisch sind oder sich zu einem numerischen Typ konvertieren lassen.
- Ein Wert eines skalaren Datentyps kann beispielsweise ohne weitere Konvertierung in einer Bedingung verwendet werden.
- Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger interpretiert oder umgekehrt ein Null-Zeiger ist äquivalent zu *false* und ein Nicht-Null-Zeiger entspricht innerhalb einer Bedingung *true*.
- Beginnend mit C++11 gibt es auch das neue Schlüsselwort **nullptr**, das anstelle der 0 für Nullzeiger verwendet werden kann.
- Ferner liegt die Nähe zwischen Zeigern und ganzen Zahlen auch in der von C++ unterstützten Adressarithmetik begründet.

- Die Spezifikation eines ganzzahligen Datentyps besteht aus einem oder mehreren Schlüsselworten, die die Größe festlegen, und dem optionalen Hinweis, ob der Datentyp vorzeichenbehaftet ist oder nicht.
- Fehlt die Angabe von **signed** oder **unsigned**, so wird grundsätzlich **signed** angenommen.
- Die einzigen Ausnahmen hiervon sind **char** und **bool**.
- Bei **char** darf der Übersetzer selbst eine Voreinstellung treffen, die sich am effizientesten auf der Zielarchitektur umsetzen lässt.

Auch wenn Angaben wie **short** oder **long** auf eine gewisse Größe hindeuten, so legt keiner der C++-Standards die damit verbundenen tatsächlichen Größen fest. Stattdessen gelten nur folgende Regeln:

- Der jeweilige „größere" Datentyp in der Reihe **char**, **short**, **int**, **long**, **long long** umfasst den Wertebereich der kleineren Datentypen, d.h. **char** ist nicht größer als **short**, **short** nicht größer als **int** usw.
- **wchar_t** basiert auf einem der anderen ganzzahligen Datentypen und übernimmt die entsprechenden Eigenschaften.
- Die korrespondierenden Datentypen mit und ohne Vorzeichen (etwa **signed int** und **unsigned int**) belegen exakt den gleichen Speicherplatz und verwenden die gleiche Zahl von Bits. (Entsprechende Konvertierungen erfolgen entsprechend der Semantik des Zweier-Komplements.)

links	Postfix-Operatoren: ++, −−, −>, ., etc
rechts	Unäre Operatoren: ++, −−, *, &, +, −, !, ~, **new**, **delete**, **sizeof**, **alignof**, **noexcept**
links	Multiplikative Operatoren: *, /, %
links	Additive Operatoren: +, −
links	Schiebe-Operatoren: <<, >>
links	Vergleichs-Operatoren: <, >, <=, >=
links	Gleichheits-Operatoren: ==, !=
links	Bitweises Und: &
links	Bitweises Exklusiv-Oder: ^
links	Bitweises Inklusiv-Oder: |
links	Logisches Und: &&
links	Logisches Oder: ||
rechts	Bedingungs-Operator: ?:
rechts	Zuweisungs-Operatoren: =, *=, /=, %=, +=, −=, >>=, <<=, &=, ^=, |=
links	Komma-Operator: ,

- Die Operatoren sind in der Reihenfolge ihres Vorrangs aufgelistet, beginnend mit den Operatoren höchster Priorität.
- Klammern können verwendet werden, um Operatoren mit Operanden auf andere Weise zu verknüpfen.
- Da nur wenige die gesamte Tabelle auswendig wissen, ist es gelegentlich ratsam, auch dann Klammern aus Gründen der Lesbarkeit einzusetzen, wenn sie nicht strikt notwendig wären.
- Sofern die Operatoren auch von C unterstützt werden, gibt es keine Änderungen der Prioritäten.

- Typ-Konvertierungen können in C sowohl implizit als auch explizit erfolgen.
- Implizite Konvertierungen werden angewendet bei Zuweisungs-Operatoren, Parameterübergaben und Operatoren. Letzteres schließt auch die unären Operatoren mit ein.
- Explizite Konvertierungen erfolgen durch die Cast-Operatoren.

Anweisungen in C++

Ausdrücke:	Ein Ausdruck, gefolgt von einem terminierenden Semikolon. Der Ausdruck wird bewertet und das Resultat nicht weiter verwendet. Typische Fälle sind Zuweisungen und Funktions- und Methodenaufrufe.
Blöcke:	Erlaubt die Zufammenfassung mehrerer Anweisungen und eröffnet einen lokalen lexikalisch begrenzten Sichtbereich.
Verzweigungen:	**if** (*condition*) *statement* **if** (*condition*) *statement* **else** *statement* **switch** (*condition*) *statement*

Wiederholungen:	**while** (*condition*) *statement*
	do *statement* **while** (*condition*);
	for (*for-init* ; *condition* ; *expression*) *statement*
	for (*for-range-declaration* : *for-range-initializer*) *statement*
Sprünge:	**return**;
	return *expression* ;
	break;, **continue**;, **goto** *identifier* ;
Ausnahmen:	**throw** *expression* ;
	try *compound-statement* *handler-seq*

- Deklarationen sind überall zulässig. Die so deklarierten Objekte sind innerhalb des umgebenden lexikalischen Blocks sichtbar, jedoch nicht vor der Deklaration.
- Im Rahmen der (später vorzustellenden) Ausnahmenbehandlungen gibt es **try**-Blöcke.
- Anweisungen können Sprungmarken vorausgehen. Da **goto**-Anweisungen eher vermieden werden, sind Sprungmarken typischerweise nur im Kontext von **switch**-Anweisungen zu sehen unter Verwendung der Schlüsselworte **case** und **default**.

MetaChars.cpp

```
#include <iostream>
using namespace std;

int main() {
   char ch;
   int meta_count(0);
   bool within_range(false);
   bool escape(false);
   while ((ch = cin.get()) != EOF) {
      // counting ...
   }
   if (meta_count == 0) {
      cout << "No";
   } else {
      cout << meta_count;
   }
   cout << " meta characters were found.\n";
} // main()
```

- **int** *meta_count*(0); ist eine Deklaration, die eine Variable namens *meta_count* anlegt, die mit 0 initialisiert wird.
- Alternativ wäre auch **int** *meta_count* = 0 korrekt gewesen. Die Notation mit den Klammern deutet aber die Initialisierung mit einem Konstruktor an.
- Ohne Initialisierungen bleibt der Wert einer lokalen Variable solange undefiniert, bis ihr explizit ein Wert zugewiesen wird.
- Zu beachten ist hier der Unterschied zwischen = (Zuweisung) und == (Vergleich).
- Innerhalb der Bedingung der **while**-Schleife wird von *cin.get*() das nächste Zeichen von der Eingabe geholt, an *ch* zugewiesen und dann mit *EOF* verglichen.

switch-Anweisung

MetaChars.cpp

```
      if (escape) {
         escape = false;
      } else {
         switch (ch) {
         case '*':
         case '?':
         case '\\':
            meta_count += 1; escape = true;
            break;
         case '[':
            meta_count += 1;
            if (!within_range) within_range += 1;
            break;
         case ']':
            if (within_range) meta_count += 1;
            within_range = 0;
            break;
         default:
            if (within_range) meta_count += 1;
            break;
         }
      }
```

MetaChars.cpp

```
switch (ch) {
case '*':
case '?':
case '\\':
   meta_count += 1; escape = true;
   break;
// cases '[' and ']' ...
default:
   if (within_range) {
      meta_count += 1;
   }
   break;
}
```

- Zu Beginn wird der Ausdruck innerhalb der **switch**-Anweisung ausgewertet.
- Dann erfolgt ein Sprung zu der Sprungmarke, die dem berechneten Wert entspricht.
- Falls keine solche Sprungmarke existiert, wird die Sprungmarke **default** ausgewählt, sofern sie existiert.

MetaChars.cpp

```
switch (ch) {
case '*':
case '?':
case '\\':
   meta_count += 1; escape = true;
   break;
// cases '[' and ']' ...
default:
   if (within_range) {
      meta_count += 1;
   }
   break;
}
```

- Beginnend von der ausgesuchten Sprungmarke wird die Ausführung bis zur nächsten *break*-Anweisung fortgesetzt oder eben bis zum Ende der **switch**-Anweisung.
- Zu beachten ist hier, dass *break*-Anweisungen jeweils nur die innerste **switch**-, **for**-, **while**- oder **do**-Anweisung verlassen.

for-Anweisung

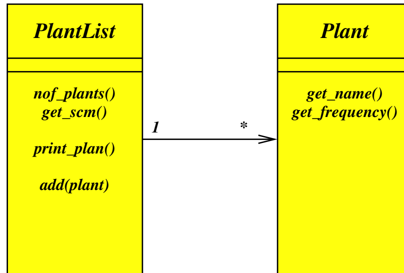
Squares.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int limit;
   cout << "Limit: "; cin >> limit;
   for (int n(1); n <= limit; n += 1) {
      cout << setw(4) << n << setw(11) << n*n << endl;
   }
}
```

- **for** (*initialization* ; *condition* ; *expression*) *statement*
  ist nahezu äquivalent zu

  ```
  {
          initialization;
          while (condition){
                  statement
                  expression;
          }
  }
  ```

- Die Initialisierung, die Bedingung und der Ausdruck dürfen leer sein. Wenn die Bedingung fehlt, wird es zur Endlosschleife.
- Der Sichtbereich von *n* wird lexikalisch auf den Bereich der **for**-Anweisung begrenzt.



- 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.

Klasse *Plant*

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
```

Klasse *Plant*

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.cpp

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

- Dies ist ein kopierender Konstruktor (*copy constructor*), der eine Initialisierung mit Hilfe eines bereits existierenden Objekts durchführt.
- Dieser Konstruktor wird in vielen Fällen implizit aufgerufen. Dazu ist beispielsweise der Fall, wenn dieser Datentyp im Rahmen der (noch vorzustellenden) Template-Klasse für Listen verwendet wird, da diese die Werte grundsätzlich kopiert.

Plant.cpp

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

- Deswegen ist dieser Konstruktor nicht nur eine Bereicherung der *Plant*-Klasse, sondern auch generell eine Voraussetzung, um Objekte dieser Klasse in Listen aufnehmen zu können.
- Wenn kein kopierender Konstruktor zur Verfügung gestellt wird (und auch sonst keine anderen Konstruktoren explizit deklariert werden) gibt es eine Voreinstellung für den kopierenden Konstruktor, der alle Felder einzeln kopiert. Dies wäre hier kein Problem, aber das kann zur bösen Falle werden, wenn Zeiger auf diese Weise vervielfältigt und dann später möglicherweise mehrfach freigegeben werden.

Plant.cpp

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

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

- Zu beachten ist hier das Schlüsselwort **const** am Ende der Signatur. Dies legt fest, dass diese Methode den (abstrakten!) Status des Objekts nicht verändert.

Klasse *PlantList*

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.h

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

// ...

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

- Dies deklariert *plants* als eine Liste von Elementen des Typs *Plant*.
- *list* ist ein Template, das einen Typparameter als Elementtyp erwartet.
- Entsprechend wird hier nicht nur *plants* deklariert, sondern auch implizit ausgehend von dem Template *list* eine neue Klasse erzeugt. Dies wird auch als Instantiierung eines Templates bezeichnet (*template instantiation*).
- Das Listen-Template gehört zur STL (*standard template library*), die Bestandteil von ISO C++ ist.

PlantList.cpp

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

PlantList::PlantList() :
   scm(0) {
} // PlantList::PlantList

int PlantList::nof_plants() const {
   return plants.size();
} // PlantList::nof_plants

int PlantList::get_scm() const {
   assert(scm > 0);
   return scm;
} // PlantList::get_scm
```

- *scm* ist dank des Konstruktors immer wohldefiniert.
- Später bestehen wir (entsprechend der Vorbedingung) darauf, dass mindestens eine Pflanze eingetragen ist, bevor ein Aufruf von *get_scm* zulässig ist.

PlantList.cpp

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

- Seit C++11 gibt es eine spezielle Form der **for**-Schleife, die die Iteration einer Datenstruktur auf elegantem Wege erlaubt.
- In jedem Schleifendurchlauf ist *plant* eine Referenz auf das aktuelle Element aus der Liste *plants*.

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);
} // PlantList::add
```

- *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"

using namespace std;

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

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

WateringPlan.cpp

```
   while (std::cin >> name && std::cin >> 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 gesamte 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 Ausführung der zugehörigen Anweisung beendet ist.

- In allen bisherigen Beispielen belegten die Objekte entweder statischen Speicherplatz oder sie lebten auf dem Stack.
- Dies vermied bislang völlig den Aufwand einer dynamischen Speicherverwaltung. Es gehört zu den Vorteilen von C++ (und einigen anderen hybriden OO-Sprachen), dass nicht für alle Objekte der Speicherplatz dynamisch belegt werden muss.

- Auf der anderen Seite ist die Beachtung einiger Richtlinien unerlässlich, wenn Klassen Zeiger auf dynamische Datenstrukturen verwenden, da
  - wegen der fehlenden automatischen Speicherbereinigung (*garbage collection*) es in der Verantwortung der Klassenimplementierung liegt, referenzierte Datenstrukturen wieder freizugeben und da
  - Konstruktoren und Zuweisungs-Operatoren per Voreinstellung nur die Datenfelder kopieren (*shallow copy*) und somit Zeigerwerte implizit vervielfältigt werden können.

Integer.hpp

```
#ifndef INTEGER_H
#define INTEGER_H

class Integer {
   public:
      // constructor
      Integer(int initval);
      // destructor
      ~Integer();
      // accessor
      int get_value() const;
      void set_value(int newval);
   private:
      int value;
}; // class Integer

#endif
```

- Die Signatur eines Destruktors besteht aus einer Tilde „∼“, dem Namen der Klasse (analog zu den Konstruktoren) und einer leeren Parameterliste.

Integer.cpp

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

using namespace std;

Integer::Integer(int intval) :
   value(intval) {
   cout << "Integer constructor: value = " <<
      value << endl;
} // Integer::Integer

Integer::~Integer() {
   cout << "Integer destructor: value = " <<
      value << endl;
} // Integer::~Integer

int Integer::get_value() const {
   return value;
} // Integer::get_value

void Integer::set_value(int newval) {
   value = newval;
} // Integer::set_value
```

- Prinzipiell können (wie in diesem Beispiel) beliebige Anweisungen wie auch Ausgaben in Destruktoren aufgenommen werden. Das kann aber zusätzliche Komplikationen mit sich bringen, wenn etwa eine Ausnahmenbehandlung in Gang gesetzt werden sollte.
- Wenn kein Destruktor angegeben wird, dann kommt eine Voreinstellung zum Zuge, die die Destruktoren für alle einzelnen Datenfelder aufruft.
- Aber auch wenn ein Destruktor spezifiziert wird, dann bleibt immer noch der automatische Aufruf der Destruktoren aller Datenfelder.
- Bei elementaren Datentypen (einschließlich den Zeigern) passiert hier jedoch nichts.

Aufrufe von Konstruktoren und Destruktoren

TestInteger1.cpp

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

using namespace std;

int main() {
   cout << "main starts" << endl;
   {
      Integer i(1);
      cout << "working on i = " << i.get_value() << endl;
   }
   cout << "main ends" << endl;
} // main
```

- Durch eine Deklaration wie hier mit *Integer i*(1) wird der passende Konstruktor aufgerufen.

```
dublin$ TestInteger1
main starts
Integer constructor: value = 1
working on i = 1
Integer destructor: value = 1
main ends
dublin$
```

- Der Sichtbereich von *i* ist statisch begrenzt auf den umgebenden Block. Die Lebenszeit beginnt und endet mit der Laufzeit des umgebenden Blocks.
- Der Destruktor wird implizit beim Verlassen des umgebenden Blocks aufgerufen.

TestInteger2.cpp

```
#include <iostream>
#include "Integer.hpp"
using namespace std;

int main() {
   cout << "main starts" << endl;

   Integer* ip (new Integer(1));
   cout << "working on ip = " << ip->get_value() << endl;
   delete ip;

   cout << "main ends" << endl;
} // main
```

- Mit *Integer∗ ip* wird *ip* als Zeiger auf *Integer* deklariert.
- Der Ausdruck **new** *Integer*(1) veranlasst das dynamische Belegen von Speicher für ein Objekt des Typs *Integer* und ruft den passenden Konstruktor auf.

```
dublin$ TestInteger2
main starts
Integer constructor: value = 1
working on ip = 1
Integer destructor: value = 1
main ends
dublin$
```

- Die Verantwortung für die Speicherfreigabe verbleibt beim Aufrufer des **new**-Operators.
- Mit **delete** *ip* wird der Destruktor von *ip* aufgerufen und danach der belegte Speicherplatz freigegeben.

TestInteger3.cpp

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

using namespace std;

int main() {
   cout << "main starts" << endl;

   Integer* ip (new Integer(1));
   Integer& ir (*ip);
   cout << "working on ip = " << ip->get_value() << endl;
   ir.set_value(2);
   cout << "working on ip = " << ip->get_value() << endl;
   delete ip;

   cout << "main ends" << endl;
} // main
```

- Mit *Integer& ir* (*∗ip*) wird *ir* als Referenz für ein *Integer*-Objekt deklariert.

- Im Vergleich zu Zeigern gibt es bei Referenzen einige Unterschiede:
  - Im Rahmen ihrer Deklaration müssen sie mit einem Objekt des referenzierten Typs verbunden werden.
  - Referenzen bleiben konstant, d.h. sie können während ihrer Lebenszeit nicht ein anderes Objekt referenzieren.
  - Referenzen werden syntaktisch wie das Objekt, das sie referenzieren, behandelt. Entsprechend wird etwa ».« an Stelle von »−>« verwendet.

ListOfFriends.hpp

```
class ListOfFriends {
   public:
      // constructor
      ListOfFriends();
      ListOfFriends(const ListOfFriends& list);
      ~ListOfFriends();

      // overloaded operators
      ListOfFriends& operator=(const ListOfFriends& list);

      // printing
      void print();

      // mutator
      void add(const Friend& f);

   private:
      struct Node* root;
      void addto(Node*& p, Node* newNode);
      void visit(const Node* const p);
}; // class ListOfFriends
```

- Ein Objekt der Klasse *ListOfFriends* verwaltet eine Liste von Freunden und ermöglicht die sortierte Ausgabe (alphabetisch nach dem Namen).
- Die Implementierung beruht auf einem sortierten binären Baum. Der Datentyp **struct** *Node* repräsentiert einen Knoten dieses Baums.
- Zu beachten ist hier, dass eine Deklaration eines Objekts des Typs **struct** *Node*∗ auch dann zulässig ist, wenn **struct** *Node* noch nicht bekannt ist, da der benötigte Speicherplatz bei Zeigern unabhängig vom referenzierten Datentyp ist.

ListOfFriends.cpp

```
struct Node {
   struct Node* left;
   struct Node* right;
   Friend f;
   Node(const Friend& newFriend);
   Node(const Node* const& node);
   ~Node();
}; // struct Node

Node::Node(const Friend& newFriend) :
   left(0), right(0), f(newFriend) {
} // Node::Node
```

- Im Vergleich zu **class** sind bei **struct** alle Komponenten implizit **public**. Da hier die Datenstruktur nur innerhalb der Implementierung deklariert wird, stört dies nicht, da sie von außen nicht einsehbar ist.
- Der hier gezeigte Konstruktor legt ein Blatt an.

ListOfFriends.cpp

```
Node::Node(const Node* const& node) :
   left(0), right(0), f(node->f) {
   if (node->left) {
      left = new Node(node->left);
   }
   if (node->right) {
      right = new Node(node->right);
   }
} // Node::Node
```

- Der zweite Konstruktor für **struct** *Node* akzeptiert einen Zeiger auf *Node* als Parameter. Die beiden **const** in der Signatur stellen sicher, dass nicht nur der (als Referenz übergebene) Zeiger nicht verändert werden darf, sondern auch nicht der Knoten, auf den dieser verweist.
- Hier ist es sinnvoll, einen Zeiger als Parameter zu übergeben, da in diesem Beispiel Knoten ausschließlich über Zeiger referenziert werden.

- Hier werden die Felder *left* und *right* zunächst in der Initialisierungssequenz auf 0 initialisiert und nachher bei Bedarf auf neu angelegte Knoten umgebogen. So ist garantiert, dass die Zeiger immer wohldefiniert sind.
- Tests wie **if** (*node*−>*left*) überprüfen, ob ein Zeiger ungleich 0 ist.
- Zu beachten ist hier, dass der Konstruktor sich selbst rekursiv für die Unterbäume *left* und *right* von *node* aufruft, sofern diese nicht 0 sind.
- Auf diese Weise erhalten wir hier eine tiefe Kopie (*deep copy*), die den gesamten Baum beginnend bei *node* dupliziert.

ListOfFriends.cpp

```
Node::~Node() {
   if (left) {
      delete left;
   }
   if (right) {
      delete right;
   }
} // Node::~Node
```

- Wie beim Konstruieren muss hier die Destruktion bei *Node* rekursiv arbeiten.
- Diese Lösung geht davon aus, dass ein Unterbaum niemals mehrfach referenziert wird.
- Nur durch die Einschränkung der Sichtbarkeit kann dies auch garantiert werden.

ListOfFriends.cpp

```
ListOfFriends::ListOfFriends() :
   root(0) {
} // ListOfFriends::ListOfFriends

ListOfFriends::ListOfFriends(const ListOfFriends& list) :
   root(0) {
   Node* r(list.root);
   if (r) {
      root = new Node (r);
   }
} // ListOfFriends::ListOfFriends
```

- Der Konstruktor ohne Parameter (*default constructor*) ist trivial: Wir setzen nur *root* auf 0.
- Der kopierende Konstruktor ist ebenso hier recht einfach, da die entscheidende Arbeit an den rekursiven Konstruktor für *Node* delegiert wird.
- Es ist hier nur darauf zu achten, dass der Konstruktor für *Node* nicht in dem Falle aufgerufen wird, wenn *list.root* gleich 0 ist.

ListOfFriends.cpp

```
ListOfFriends::~ListOfFriends() {
   if (root) {
      delete root;
   }
} // ListOfFriends::~ListOfFriends
```

- Analog delegiert der Destruktor für *ListOfFriends* die Arbeit an den Destruktor für *Node*.
- Es ist nicht schlimm, wenn der **delete**-Operator für 0-Zeiger aufgerufen wird. Das wird vom ISO-Standard für C++ ausdrücklich erlaubt. Die **if**-Anweisung spart aber Ausführungszeit.

ListOfFriends.cpp

```
ListOfFriends& ListOfFriends::operator=
      (const ListOfFriends& list) {
   if (this != &list) { // protect against self-assignment
      if (root) {
         delete root;
      }
      if (list.root) {
         root = new Node (list.root);
      } else {
         root = nullptr;
      }
   }
   return *this;
} // ListOfFriends::operator=
```

- Ein rekursiv arbeitender kopierender Konstruktur und zugehöriger Destruktor genügen alleine nicht, da der voreingestellte Zuweisungs-Operator nur den Wurzelzeiger kopieren würde (*shallow copy*) und eine rekursive Kopie (*deep copy*) unterbleiben würde.

- Dies würde die wichtige Annahme (des Destruktors) verletzen, dass der selbe Baum nicht von mehreren Objekten des Typs *ListOfFriends* referenziert werden darf.
- Entsprechend ist die Implementierung unvollständig, solange eine simple Zuweisung von *ListOfFriends*-Objekten diese wichtige Annahme verletzen kann.
- Bei der Implementierung des Zuweisungs-Operators ist darauf zu achten, dass Objekte an sich selbst zugewiesen werden können. **this** repräsentiert einen Zeiger auf das Objekt, auf der die aufgerufene Methode arbeitet. &*list* ermittelt die Adresse von *list* und erlaubt somit einen Vergleich von Zeigerwerten.

ListOfFriends.cpp

```
void ListOfFriends::addto(Node*& p, Node* newNode) {
   if (p) {
      if (newNode->f.get_name() < p->f.get_name()) {
         addto(p->left, newNode);
      } else {
         addto(p->right, newNode);
      }
   } else {
      p = newNode;
   }
} // ListOfFriends::addto

void ListOfFriends::add(const Friend& f) {
   Node* node( new Node(f) );
   addto(root, node);
} // ListOfFriends::add
```

- Wenn ein neuer Freund in die Liste aufgenommen wird, ist ein neues Blatt anzulegen, das auf rekursive Weise in den Baum mit Hilfe der privaten Methode *addto* eingefügt wird.

ListOfFriends.cpp

```
void ListOfFriends::visit(const Node* const p) {
   if (p) {
      visit(p->left);
      cout << p->f.get_name() << ": " <<
         p->f.get_info() << endl;
      visit(p->right);
   }
} // ListOfFriends::visit

void ListOfFriends::print() {
   visit(root);
} // ListOfFriends::print
```

- Analog erfolgt die Ausgabe rekursiv mit Hilfe der privaten Methode *visit*.

TestFriends.cpp

```
   ListOfFriends list1;
```

- Diese Deklaration ruft implizit den Konstruktor von *ListOfFriends* auf, der keine Parameter verlangt (*default constructor*). In diesem Falle wird *root* einfach auf 0 gesetzt werden.

TestFriends.cpp

```
   ListOfFriends list2(list1);
```

- Diese Deklaration führt zum Aufruf des kopierenden Konstruktors, der den vollständigen Baum von *list1* für *list2* dupliziert.

TestFriends.cpp

```
   ListOfFriends list3;
   list3 = list1;
```

- Hier wird zunächst der Konstruktor von *ListOfFriends* ohne Parameter aufgerufen (*default constructor*).
- Danach kommt es zur Ausführung des Zuweisungs-Operators, der den Baum von *list1* dupliziert und bei *list3* einhängt.

Function.hpp

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual std::string get_name() const = 0;
      virtual double execute(double x) const = 0;
}; // class Function
```

- Polymorphe Methoden einer Basis-Klasse können in einer abgeleiteten Klasse überdefiniert werden.
- Eine Methode wird durch das Schlüsselwort **virtual** als polymorph gekennzeichnet.
- Dies wird auch als *dynamischer Polymorphismus* bezeichnet, da die auszuführende Methode zur Laufzeit bestimmt wird,

Function.hpp

```
     virtual std::string get_name() const = 0;
```

- Die Angabe von = 0 am Ende einer Signatur einer polymorphen Methode ermöglicht den Verzicht auf eine zugehörige Implementierung.
- In diesem Falle gibt es nur Implementierungen in abgeleiteten Klassen und nicht in der Basis-Klasse.
- So gekennzeichnete Methoden werden *abstrakt* genannt.
- Klassen mit mindestens einer solchen Methode werden *abstrakte Klassen* genannt.
- Abstrakte Klassen können nicht instantiiert werden.

Function.hpp

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual std::string get_name() const = 0;
      virtual double execute(double x) const = 0;
}; // class Function
```

- Wenn wie in diesem Beispiel alle Methoden abstrakt sind (oder wie beim Dekonstruktor innerhalb der Klassendeklaration implementiert werden), kann die zugehörige Implementierung vollständig entfallen. Entsprechend gibt es keine zugehörige Datei namens *Function.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 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"

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

double Sinus::execute(double x) const {
   return std::sin(x);
} // Sinus::execute
```

- Alle Methoden, die nicht abstrakt sind und nicht in einer der Basisklassen definiert worden sind, müssen implementiert werden.
- Hier wird auf die Definition eines Dekonstruktors verzichtet. Stattdessen kommt der leere Dekonstruktor der Basisklasse zum Zuge.

Die Benutzung einer polymorphen Klasse

TestSinus.cpp

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

using namespace std;

int main() {
   Function* f(new Sinus());
   double x;

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

- Variablen des Typs *Function* können nicht deklariert werden, weil *Function* eine abstrakte Klasse ist.
- Stattdessen ist es aber zulässig, Zeiger oder Referenzen auf *Function* zu deklarieren, also *Function∗* oder *Function&*.

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!

Die Benutzung einer polymorphen Klasse

TestSinus.cpp

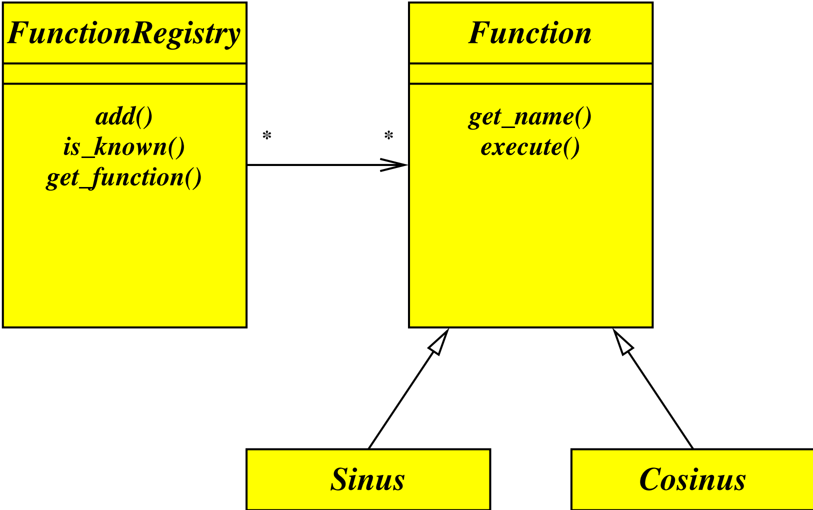
```
#include <iostream>
#include "Sinus.hpp"

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
```

- Wenn eine Methode mit dem Schlüsselwort **virtual** versehen ist, dann erfolgt die Bestimmung der zugeordneten Methodenimplementierung *erst zur Laufzeit* in Abhängigkeit vom dynamischen Typ, der bei Zeigern und Referenzen eine beliebige Erweiterung des deklarierten Typs sein kann.

TestSinus.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 der dynamische Typ hier *Sinus*$*$ ist.


FunctionRegistry
add()
is_known()
get_function()
*
*
Function
get_name()
execute()
Sinus
Cosinus

- Die Einführung einer Klasse *FunctionRegistry* erlaubt es, Funktionen über ihren Namen auszuwählen.
- Hiermit ist es beispielsweise möglich, den Namen einer Funktion einzulesen und dann mit dem gegebenen Namen ein zugehöriges Funktionsobjekt zu erhalten.
- Dank der Kompatibilität einer abgeleiteten Klasse zu den Basisklassen ist es möglich, heterogene Listen (d.h. Listen mit Objekten unterschiedlicher Typen) zu verwalten, sofern eine gemeinsame Basisklasse zur Verfügung steht. In diesem Beispiel ist das *Function*.

FunctionRegistry.hpp

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

class FunctionRegistry {
   public:
      void add(Function* f);
      bool is_known(std::string fname) const;
      Function* get_function(std::string fname);
   private:
      std::map< std::string, Function* > registry;
}; // class FunctionRegistry
```

- *map* ist eine Implementierung für assoziative Arrays und gehört zu den generischen Klassen der Standard-Template-Library (STL)
- *map* erwartet zwei Typen als Parameter: den 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(std::string fname) const {
   return registry.find(fname) != registry.end();
} // FunctionRegistry::is_known

Function* FunctionRegistry::get_function(std::string fname) {
   return registry[fname];
} // FunctionRegistry::get_function
```

- Instantiierungen der generischen Klasse *map* können analog zu regulären Arrays verwendet werden, da der []-Operator für sie überladen wurde.
- *registry.find* liefert einen Iterator, der auf *registry.end* verweist, falls der gegebene Index bislang noch nicht belegt wurde.

FunctionRegistry.cpp

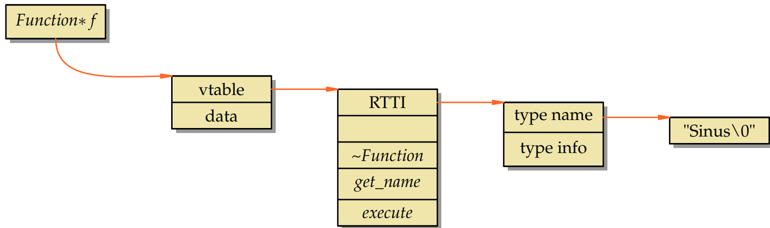
```
bool FunctionRegistry::is_known(std::string fname) const {
   return registry.find(fname) != registry.end();
} // FunctionRegistry::is_known
```

- 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 das gewünschte Objekt, sondern einen Iterator darauf.
- 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
```

Wie teuer ist dynamischer Polymorphismus?



- Nicht-polymorphe Methoden und reguläre Funktionen können in C++ direkt aufgerufen werden, d.h. die Sprungadresse ist direkt im Maschinen-Code verankert.
- Bei polymorphen Methoden muss zunächst hinter dem Objektzeiger, der sogenannte *vtable*-Zeiger geladen werden, hinter dem sich wiederum eine Liste mit Funktionszeigern zu den einzelnen Methoden verbirgt.
- Die Kosten einer polymorphen Methode belaufen sich entsprechend auf zwei nicht parallelisierbare Speicherzugriffe. Im ungünstigsten Falle (d.h. nichts davon ist im Cache) kostet dies bei aktuellen Systemen ca. 200 ns.

- Da der Aufruf polymorpher Methoden (also solcher Methoden, die mit **virtual** ausgezeichnet sind) zusätzliche Kosten während der Laufzeit verursacht, stellt sich die Frage, wann dieser Aufwand gerechtfertigt ist.
- Sinnvoll ist dynamischer Polymorphismus insbesondere, wenn
  - Container mit Zeiger oder Referenzen auf heterogene Objekte gefüllt werden, die alle eine Basisklasse gemeinsam haben oder
  - unbekannte Erweiterungen einer Basisklasse erst zur Laufzeit geladen werden.

```
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.

- Die bisher zu C++ erschienen ISO-Standards (bis einschließlich ISO 14882-2012) sehen das dynamische Laden von Klassen nicht vor.
- Der POSIX-Standard (IEEE Standard 1003.1) schließt einige C-Funktionen ein, die das dynamische Nachladen von speziell übersetzten Modulen (*shared objects*) ermöglichen.
- Diese Schnittstelle kann auch von C++ aus genutzt werden, da grundsätzlich C-Funktionen auch von C++ aus verwendbar sind.
- Es sind hierbei allerdings Feinheiten zu beachten, da wegen des Überladens in C++ Symbolnamen auf der Ebene des Laders nicht mehr mit den in C++ verwendeten Namen übereinstimmen. Erschwerend kommt hinzu, dass die Abbildung von Namen in C++ in Symbolnamen – das sogenannte *name mangling* – nicht standardisiert ist.

```
#include <dlfcn.h>
#include <link.h>

void* dlopen(const char* pathname, int mode);
char* dlerror(void);
```

- *dlopen* lädt ein Modul (*shared object*, typischerweise mit der Dateiendung „.so"), dessen Dateiname bei *pathname* spezifiziert wird.
- Der Parameter *mode* legt zwei Punkte unabhängig voneinander fest:
  - Wann werden die Symbole aufgelöst? Entweder sofort (*RTLD_NOW*) oder so spät wie möglich (*RTLD_LAZY*). Letzteres wird normalerweise bevorzugt.
  - Sind die geladenen globalen Symbole für später zu ladende Module sichtbar (*RTLD_GLOBAL*) oder wird ihre Sichtbarkeit lokal begrenzt (*RTLD_LOCAL*)? Hier wird zur Vermeidung von Konflikten typischerweise *RTLD_LOCAL* gewählt.
- Wenn das Laden nicht klappt, dann kann *dlerror* aufgerufen werden, um eine passende Fehlermeldung abzurufen.

```
#include <dlfcn.h>

void* dlsym(void* restrict handle, const char* restrict name);
int dlclose(void* handle);
```

- Die Funktion *dlsym* erlaubt es, Symbolnamen in Adressen zu konvertieren. Im Falle von Funktionen lässt sich auf diese Weise ein Funktionszeiger gewinnen. Zu beachten ist hier, dass nur bei C-Funktionen davon ausgegangen werden kann, dass der C-Funktionsname dem Symbolnamen entspricht. Bei C++ ist das ausgeschlossen. Als *handle* wird der **return**-Wert von *dlopen* verwendet, *name* ist der Symbolname.
- Mit *dlclose* kann ein nicht mehr benötigtes Modul wieder entfernt werden.

Definition einer C-Funktion in C++


```
extern "C" void do_something() {
   // beliebiger C++-Programmtext
}
```

- In C++ kann eine Funktion mit **extern** "C" ausgezeichnet werden.
- Diese Funktion ist dann von C aus unter ihrem Namen aufrufbar.
- Ein Überladen solcher Funktionen ist naturgemäß nicht möglich, da C dies nicht unterstützt.
- Innerhalb dieser Funktion sind allerdings beliebige C++-Konstrukte möglich.
- Ein solche C-Funktion kann benutzt werden, um ein Objekt der C++-Klasse zu konstruieren oder ein Objekt einer passenden Factory-Klasse zu erzeugen, mit der Objekte der eigentlichen Klasse konstruiert werden können.

Sinus.cpp

```
extern "C" Function* construct() {
   return new Sinus();
}
```

- Im Falle sogenannter Singleton-Objekte (d.h. Fälle, bei denen typischerweise pro Klasse nur ein Objekt erzeugt wird), genügt eine einfache Konstruktor-Funktion.
- Diese darf sogar einen global nicht eindeutigen Namen tragen – vorausgesetzt, wir laden das Modul mit der Option *RTLD_LOCAL*. Dann ist das entsprechende Symbol nur über den von *dlopen* zurückgelieferten Zeiger in Verbindung mit der *dlsym*-Funktion zugänglich.

DynFunctionRegistry.hpp

```
class DynFunctionRegistry {
   public:
      // constructors
      DynFunctionRegistry();
      DynFunctionRegistry(const std::string& dirname);

      void add(Function* f);
      bool is_known(const std::string& fname);
      Function* get_function(const std::string& fname);
   private:
      const std::string dir;
      std::map< std::string, Function* > registry;
      Function* dynload(const std::string& fname);
}; // class DynFunctionRegistry
```

- Neben dem Default-Konstruktor gibt es jetzt einen weiteren, der einen Verzeichnisnamen erhält, in dem die zu ladenden Module gesucht werden.
- Ferner kommt noch die private Methode *dynload* hinzu, deren Aufgabe es ist, ein Modul, das die angegebene Funktion implementiert, dynamisch nachzuladen und ein entsprechendes Singleton-Objekt zu erzeugen.

DynFunctionRegistry.cpp

```
typedef Function* FunctionConstructor();

Function* DynFunctionRegistry::dynload(const std::string& name) {
   std::string path(dir);
   if (path.size() > 0) path += "/";
   path += name; path += ".so";
   void* handle = dlopen(path.c_str(), RTLD_LAZY | RTLD_LOCAL);
   if (!handle) return 0;
   FunctionConstructor* constructor =
      (FunctionConstructor*) dlsym(handle, "construct");
   if (!constructor) {
      dlclose(handle); return 0;
   }
   return constructor();
}
```

- Zunächst wird aus *name* ein Pfad bestimmt, unter der das passende Modul abgelegt sein könnte.
- Dann wird mit *dlopen* versucht, es zu laden.
- Wenn dies erfolgreich war, wird mit Hilfe von *dlsym* die Adresse der *construct*-Funktion ermittelt und diese im Erfolgsfalle aufgerufen.

Abrufen einer Funktion

DynFunctionRegistry.cpp

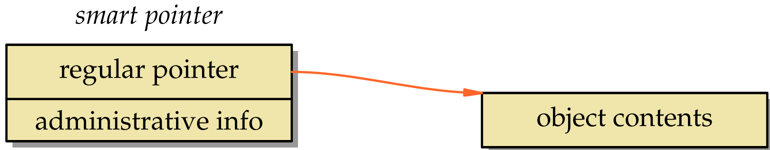
```
Function* DynFunctionRegistry::get_function(const std::string& fname) {
   auto it(registry.find(fname));
   Function* f;
   if (it == registry.end()) {
      f = dynload(fname);
      if (f) {
         add(f);
         if (f->get_name() != fname) registry[fname] = f;
      }
   } else {
      f = it->second;
   }
   return f;
} // FunctionRegistry::get_function
```

- Innerhalb der *map*-Template-Klasse gibt es ebenfalls einen *iterator*-Typ, der hier mit dem Resultat von *find* initialisiert wird.
- Wenn dieser Iterator dereferenziert wird, liefert ein Paar mit den Komponenten *first* (Index) und *second* (eigentlicher Wert hinter dem Index).
- Falls der Name bislang nicht eingetragen ist, wird mit Hilfe von *dynload* versucht, das zugehörige Modul dynamisch nachzuladen.

DynFunctionRegistry.cpp

```
auto it(registry.find(fname));
```

- Beginnend mit C++11 kann bei einer Deklaration auf die Spezifikation eines Typs mit Hilfe des Schlüsselworts **auto** verzichtet werden, wenn sich der gewünschte Typ von der Initialisierung ableiten lässt.
- In diesem Beispiel muss nicht der lange Typname *std*::*map*< *std*::*string*, *Function*∗ >::*iterator* hingeschrieben werden, weil der Übersetzer das selbst automatisiert von dem Rückgabetyp von *registry*.*find*() ableiten kann.



- Intelligente Zeiger (*smart pointers*) entsprechend 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.

- 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 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.

- 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 entsprechend 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.

- 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.

Lokale Zeiger: Traditionelle Technik

ptrex.cpp

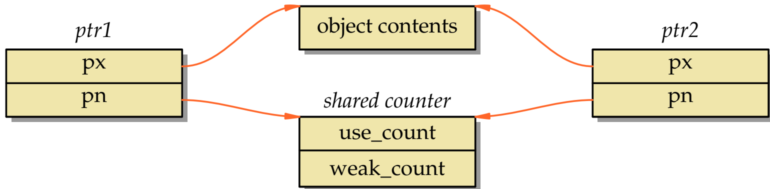
```
void f(int i) {
   Object* ptr = new Object(i);
   if (i == 2) {
      throw something();
   }
   delete ptr;
}
```

- 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.

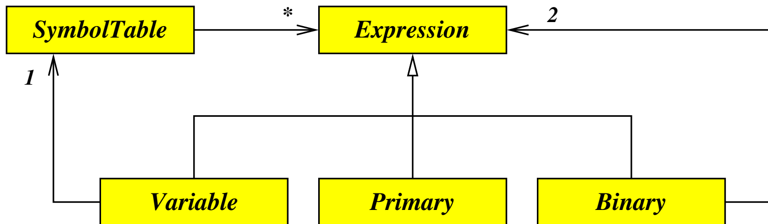
ptrex2.cpp

```
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.



- 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.



- Für einen Taschenrechner haben wir eine Datenstruktur für Syntaxbäume (*Expression*) mit den abgeleiteten Klassen *Variable*, *Primary* und *Binary*.
- Einträge in der Symboltabelle können auch auf Syntaxbäume verweisen.

expression.hpp

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

typedef std::shared_ptr<Expression> ExpressionPtr;
```

- 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);
      virtual Value evaluate() const;
   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);
      virtual Value evaluate() const;
      void set(ExpressionPtr expr);
   private:
      SymbolTable& symtab;
      std::string varname;
};
typedef std::shared_ptr<Variable> VariablePtr;
```

- 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.

Intelligente Zeiger auf neuerzeugte Objekte

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 sicherzustellen, dass es bei einem Zählerobjekt bleibt.
- 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.

list.hpp

```
template <typename T>
class List {
   private:
      struct Element;
      typedef std::shared_ptr<Element> Link;
      typedef std::weak_ptr<Element> WeakLink;
      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.

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;
};
```

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.

Vergleich von schwachen Zeigern

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;
typedef std::shared_ptr<Object> ObjectPtr;
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;
typedef std::shared_ptr<Object> ObjectPtr;
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.