1 Allgemeines

- Java ist eine aus C++ heraus entstandene Sprache. D.h. zunächst, daß die Syntax sehr ähnlich ist, die Kontrollstrukturen (`if`, `if-else`, `while`, `do-while`, `for`, `switch-case-default`, `break`, `continue`, `return`) und die Operatoren (`=`, `==`, `!=`, `++`, `--`, ...) sind die gleichen.
  Auch die Kommentar-Zeichen (`//` und `/* ... */`) sind die gleichen. `/** ... */` existiert nicht, bzw. hat keine besondere Bedeutung und wird wie `/* ... */` behandelt.
- Erste Unterschiede ergeben sich im Programmaufbau. C++ ist eine auf C aufbauende Sprache und bleibt dieser in vielen Punkten treu. So benötigen C++-Programme ebenso wie C-Programme eine globale `main`-Funktion, die beim Programmstart ausgeführt wird.
  Beispiel:

  ```
  Funktions- und Klassen-Definitionen
  main-Funktion
  ```

  Die `main`-Funktion kann zwei verschiedene Formen haben:

  ```
  int main () {}
  ```
  oder
  ```
  int main (int argc, char *argv[]) {}
  ```

  `argv` enthält die beim Programmaufruf angegebenen Kommandozeilenparameter. `argv[0]` ist der Programmname, `argv[1]` der erste Parameter und so weiter. `argc` gibt die Anzahl der Elemente im `argv`-Feld an.
- In C++ wird unterschieden zwischen „Deklaration“ und „Definition“. Im allgemeinen dienen Deklarationen der „Bekanntmachung“ von Funktionen. Definitionen dagegen sind Code erzeugend. (Ausnahme sind Variablen, sie werden durch eine Deklaration erzeugt. Durch eine Deklaration mit dem Schlüsselwort „extern“ werden sie nur bekannt gemacht (siehe 4.2).)
- Zur Strukturierung von Programmen, werden Programmteile in verschiedenen Dateien gespeichert (Sourcefiles-Endung ‚.cc‘) und auch getrennt compiliert (➪ ergibt die Objectfiles-Endung ‚.o‘). Anschließend werden die Objectfiles zu einem ausführbaren Programm (Executable) „gelinkt“.
- Soll ein Programmteil beispielsweise auf Klassen zugreifen, die in einer anderen Datei definiert sind, so müssen ihm diese bekannt gemacht (deklariert) werden. Dazu werden die Deklarationen der Klassen, Funktionen, Variablen, etc. in sogenannten „Headerfiles“‘ (Endung .hh) zusammengefasst. Diese Deklarationen sind nicht Code erzeugend und für alle einsehbar. Die zugehörigen Definitionen werden in den Sourcefiles (.cc) gespeichert und bleiben den anderen Programmteilen verborgen.
- Wichtig ist, daß in den „Headerfiles“ keine(!) Definitionen (Code erzeugende Programmteile) enthalten sein dürfen, da diese sonst mehrfach in verschiedenen Programmteilen (Objectfiles) auftauchen und später beim Linken zu Mehrdeutigkeiten führen! (Ausnahme: „Inline“-Funktionen.)
- Üblich ist es dabei pro Klasse eine Datei (eigentlich zwei: ein Source- und ein Headerfile) anzulegen. Auch für die `main`-Funktion wird eine Datei angelegt.
- Ein C++-Programm wird in drei Phasen übersetzt. Zunächst werden die Programmteile durch den Präprozessor geschickt, dann compiliert und anschließend zusammengelinkt. Der Präprozessor wer-

tet die sogenannten „Präprozessor-Direktiven“ aus. Präprozessor-Direktiven sind durch ein vorangestelltes ‚#‘ gekennzeichnet. Die wichtigsten sind:
`#include "Dateiname"` - bindet die angegebene Datei (aus dem Arbeitsverzeichnis) ein.
`#include <Dateiname>` - bindet die angegebene Datei (aus den Suchpfaden) ein.
`#define Variablenname Definition` - belegt den Variablennamen mit dem Definitionsstring. Im folgenden wird jedes Vorkommen von „Variablenname“ textuell durch „Definition“ ersetzt.
`#ifdef Variablenname` - testet, ob eine Variable dieses Namens schon definiert wurde. Falls nicht, wird der Code bis zum folgenden `#endif` entfernt.
`#ifndef Variablenname` - testet, ob eine Variable dieses Namens schon definiert wurde. Falls ja, wird der Code bis zum folgenden `#endif` entfernt.
`#endif` - s. `#ifdef` bzw. `#ifndef`

Beispiel:

Datei: Header.hh
```
#ifndef _HEADER_
#define _HEADER_
Zeile 1
#define blablabla Zeile 2
#endif
```

Datei: Source.cc
```
#include "Header.hh"
#include "Header.hh"
blablabla
Zeile 3
```

Wird nun Source.cc compiliert, so wird zunächst der Präprozessor gestartet. Anschließend liegt folgender Text vor, der dem eigentlichen Compiler übergeben wird:

```
Zeile 1
Zeile 2
Zeile 3
```

Zunächst wurde der Text aus Header.hh eingebunden. Da die Variable `_HEADER_` noch nicht existiert, bleibt der folgende Code bis zum `#endif` erhalten. Dadurch werden die Variablen `_HEADER_` und `blablabla` definiert und die ‚`Zeile 1`‘ eingebunden. Anschließend wird erneut Header.hh geladen, da aber `_HEADER_` schon definiert ist, wird der folgende Teile bis zum `#endif` entfernt. Diese `#ifndef`-`#define`-`#endif`-Klammerung von Headerfiles ist üblich (und meist auch notwendig) um eine „Redeklaration“ zu vermeiden, wenn eine Datei mehrfach eingebunden wird (lässt sich oft nicht vermeiden). Anschließend wird das `blablabla` durch ‚`Zeile 2`‘ ersetzt und dann findet der Präprozessor keine weiteren Direktiven und auch nichts, was noch ersetzt werden könnte (keine weiteren `blablabla`‘s).

2 Wie man ein C++-Programm übersetzt

Für das Compilieren unter Solaris ist es notwendig das Paket ‚spro' in die Umgebungsvariable „RCINFO_ILIST" mitaufzunehmen. (Anschließend einmal neu einloggen!) Dadurch werden die eigenen Suchpfade so erweitert, daß der C++-Compiler ‚`CC`' gefunden wird. Tut man dies nicht, so muß man den Compiler direkt über ‚`/opt/SUNWspro/bin/CC`' aufrufen.

2.1 Direktes Erstellen eines ausführbaren Programms

Soll aus einer Sourcedatei (.cc) direkt ein ausführbares Programm generiert werden, so muß diese Datei die Definition einer globalen `main`-Funktion beinhalten. Compiliert wird dann mit der Anweisung:

```
CC <Dateiname>
```

In diesem Fall wird ein Executable namens `a.out` erzeugt. Soll es einen anderen Namen bekommen, so kann man schreiben:

```
CC <Dateiname> -o <Executablename>
```

2.2 Erstellen eines Objectfiles

Ist ein Programm über mehrere Dateien verteilt, so ist es sinnvoll, diese zunächst als Objectfiles (compiliert, aber nicht ausführbar) zu übersetzen. Ändert sich eine Datei, so muß dann nur diese neu übersetzt werden. Ein Objectfile erzeugt man mit der Anweisung:

```
CC -c <Dateiname>
```

Das Objectfile heißt dann wie die Datei, nur daß es die Endung ‚.o' trägt. Hat man aus allen Programmteilen Objectfiles erzeugt, so können sie zu einem Executable gelinkt werden. Wichtig ist nur, daß in einem der Objectfiles die globale `main`-Funktion definiert ist. Linken kann man folgendermaßen:

```
CC <Objectfile> <Objectfile> <...> ... -o <Executablename>
```

Fehlt die Option ‚`-o <Executablename>`', so wird eine Datei namens `a.out` erzeugt.

2.3 Makefiles

In unserem Seminar braucht ihr die Programmteile nicht direkt übersetzen. Dafür gibt es ein Programm names „make". Dieses wertet ein sog. „Makefile" aus, eine Datei in der alle notwendigen Angaben für die Compilierung auch komplexer Programme gemacht werden können. Dazu werden dort z.B. sämtliche Programmteile, Suchpfade, benötigte andere Bibliotheken, etc. angegeben. Mit dem einfachen Kommando „make" wird dann das eigene Programm übersetzt. Wie das genau für eure Agenten in unserem Seminar funktioniert, wird noch erklärt.

3 Fundamentale Datentypen

Die fundamentalen Datentypen in C++ sind: `void`, `char`, `int`, `float`, `double`, `enum`, (`bool`)

Diese können noch durch die folgenden Schlüsselwörter modifiziert werden: `short`, `long`, `unsigned`, `signed`, (`const`)

4 Variablen

4.1 Variablen-Deklaration

Wie bereits erwähnt sind Variablen-Deklarationen codeerzeugend. Es wird also bei einer Deklaration der entsprechende Speicherplatz für eine Variable bereitgestellt. Eine Deklaration hat (wie in Java) die Form:

```
Typname Variablenname{, Variablenname};
```

4.2 Extern-Deklaration von Variablen

Soll eine Variable nur bekannt gemacht werden, aber für sie kein Speicher angelegt werden (weil das an anderer Stelle schon geschehen ist), so wird sie als „extern“ deklariert:

```
extern Typname Variablenname;
```

4.3 Initialisierung

Variablen können gleich bei ihrer Deklaration initialisiert werden. Dies geschieht folgendermaßen:

```
Typname Variablenname = Wert;
```

oder:

```
Typname Variablenname (Wert{, Wert});
```

bei mehrparametrigen Konstruktoren.

4.4 Pointerdatentyen

Für jeden Datentypen in C++ gibt es einen entsprechenden Pointertypen. Dieser lautet: `Typname*`.
Die Bedeutung einer Variablen von einem Pointertyp ist, daß an der in der Variablen angegebenen Speicheradresse ein Wert des entsprechenden Typs liegt.
Pointervariablen werden folgendermaßen deklariert:

```
Typname* Variablenname;
```

Deklarationen von Pointervariablen stellen Speicher für eine Adressangabe bereit, nicht für den Wert des angegebenen Typs. Eine Pointervariable enthält (wie schon gesagt) eine Speicheradresse. Auf den Inhalt dieser Adresse referenziert man `*Variablenname`.
Pointervariablen können auch undefiniert sein. Sie haben dann den Wert `NULL` (0).
Man kann auch an die Adresse einer „normal“ deklarierten Variablen mittels des sog. „Adressoperators“ & kommen: `&Variablenname`. Beispiel:

```
int a = 1, b = 1;                    // a und b haben den Wert 1
int* p = &a;                         // Zeiger p enthält Adresse von a
*p = 2;                              // a hat den Wert 2
```

```
p = &b;                          // p enthält die Adresse von b
*p = 3;                          // b hat den Wert 3
```

4.5 Referenzdatentypen

Man kann in C++ nicht nur über Pointer mehrfach auf dieselbe Variable referenzieren. Eine weitere Möglichkeit bieten „Referenzvariablen“, welche in jedem Fall (wenn möglich) den Pointervariablen vorzuziehen sind! Referenzvariablen werden folgendermaßen deklariert:

```
Typname &Referenzvariablenname = Variablenname;
```

Wichtig ist hierbei die Initialisierung: Referenzvariablen können (im Gegensatz zu Pointervariablen) **nicht** undefiniert sein! Eine Referenzvariable ist für ihre Lebensdauer auch genau an diese eine Variable gebunden und kann nicht auf eine andere Variable referenzieren. Der Typ der Variablen lautet: `Typname&`.
Beispiel:

```
int a = 1, b = 1;                // a und b haben den Wert 1
int &p = a;                      // p ist Referenzvariable für a
p = 2;                           // a hat den Wert 2;
p = b;                           // a hat wieder den Wert 1 (von b)!
p = 3;                           // a hat den Wert 3
```

4.6 Konstanten

Konstanten werden wie Variablen deklariert und mit dem Schlüsselwort `const` versehen. Es gibt dabei verschiedene Varianten. Eine einfache Konstante wird wie folgt deklariert:

```
const Typ Variablenname;
```

Einen Zeiger auf einen konstanten Wert deklariert man:

```
const Typ* Variablenname;
```

Soll der Zeiger selbst, nicht aber der referenzierte Wert konstant sein schreibt man:

```
Typ* const Variablenname;
```

Sollen Zeiger und Wert konstant sein, kann man das folgendermaßen beschreiben:

```
const Typ* const Variablenname;
```

Auch Referenzen können als konstant deklariert werden:

```
const Typ &Variablenname;
```

Häufig ist es auch wichtig (nicht-statische) Elementfunktionen von Klassen als konstant deklarieren zu können. In diesem Fall kann man sie nämlich auch für konstante Objekte (Instanzen der Klasse) noch aufrufen. Konstante Elementfunktionen werden folgendermaßen deklariert:

```
Typ Funktionsname (...) const;
```

4.7 Variablenfelder (Arrays)

Arrays werden in C (gibt es also in C++) folgendermaßen deklariert:

```
Typname Variablenname[Feldgröße];
```

Arrays sind indiziert von 0 bis Feldgröße-1. Anders als bei Java kann die Feldgröße des Arrays nicht abgefragt werden. Feldüberschreitungen erzeugen auch keine Exception, sondern fuhrwerken munter im Speicher rum. Bestenfalls stürzt das Programm mit einer Fehlermeldung ‚segmentation violation' oder auch mal ‚bus error' ab. Aus diesem Grund hat man in C++ angefangen, eigene „Container"-Klassen für z.B. Arrays zu definieren, die ebenso sicher und mächtig sind wie z.B. die in Java. Mittlerweile sind welche in den Sprachstandard übernommen worden (siehe unten) und sollten auch benutzt werden!.
Bei den „alten" (C-)Arrays gelten folgende Äquivalenzen (`Variablenname` ist vom Typ `Typname*`):

`Variablenname` und `&Variablenname[0]` (Typ: `Typname*`)
`*Variablenname` und `Variablenname[0]` (Typ: `Typname`)
`Variablenname+1` und `&Variablenname[1]` (Typ: `Typname*`)
`*(Variablenname+1)` und `Variablenname[1]` (Typ: `Typname`)
usw.

4.8 Dynamische Variablenallokation

Variablen können in C++ dynamisch über den Operator:

```
Typname* Adresse = new Typname;
```

erzeugt werden. Der `new`-Operator reserviert Speicherplatz für einen Wert des angegebenen Typs und liefert einen Zeiger auf diesen Speicherplatz zurück (Typ: `Typname*`). Für das Freigeben dieses reservierten Speichers ist in C++ der Programmierer selbst verantwortlich!!! Dies geschieht über den Operator:

```
delete Adresse;
```

Speicherplatz für Arrays wird mittels:

```
Typname* Adresse = new Typname[Feldgröße];
```

reserviert und durch:

```
delete [] Adresse;
```

wieder freigegeben. Auch das Allozieren von Arrays liefert einen Pointer auf den entsprechenden Typ, und zwar auf das erste Element des Felds. Mit `Adresse[0]`, `Adresse[1]`, ... kann dann auf die einzelnen Elemente zugegriffen werden.

5 Funktionen

5.1 Funktions-Deklaration

Funktionen werden folgendermaßen deklariert:

```
Typname Funktionsname(Typname Variablenname, Typname Variablenname, ...);
```

5.2 Funktions-Definition

Die Definition einer Funktion sieht ähnlich der Deklaration aus:

```
Typname Funktionsname(Typname Variablenname, Typname Variablenname, ...)
{
  Anweisungen
}
```

Wurde eine Funktion definiert, muß sie nicht noch zusätzlich deklariert werden. Sie ist dann schon bekannt.
Eine Funktion ist (wie in Java) durch ihren Funktionsnamen nicht eindeutig bestimmt, sondern durch ihren Funktionsnamen und ihre Parameterliste. Dadurch sind polymorphe Funktionen möglich. Auch Defaultwerte für Parameter sind wie in Java möglich und können bei der Deklaration angegeben werden.

6 Klassen

Die zentrale Datenstruktur in C++ ist die Klasse. Diese kapselt eine gewisse Funktionalität und definiert über öffentlich aufrufbare Funktionen und zugängliche Variablen eine Schnittstelle nach Aussen.

6.1 Klassen-Deklaration

Die einfachste Form einer Klassendeklaration hat die Form:

```
class Klassenname
{
  Deklarationen der Klassenkomponenten
};
```

Soll die Klasse von einer oder mehreren anderen Klassen abgeleitet werden, so hat die Deklaration die folgende Form:

```
class Klassenname: public Klassenname, public Klassenname, ...
{
  Deklaration der Klassenkomponenten
};
```

Klassenkomponenten können wie in Java *public*, *protected* oder *private* sein. Dies muß allerdings nicht für jede Komponente einzeln deklariert werden, sondern kann sich immer auf ganze Abschnitte beziehen:

```
class Klassenname
{
```

```
  public:
    Deklarationen
  protected:
    Deklarationen
  public:
    Deklarationen
  private:
    Deklarationen
};
```

Klassenbestandteile können Member-Funktionen (s. Deklaration von Funktionen), Variablen (s. Deklaration von Variablen), statische Funktionen, statische Variablen und natürlich Konstruktoren und ein Destruktor sein; (fast) alles wie in Java. Statische Komponenten werden deklariert, indem das Schlüsselwort „`static`“ vorangestellt wird.

6.2 Konstruktoren

Konstruktoren werden ähnlich wie in Java deklariert. Sie sind Funktionen, die wie die Klasse heissen und keinen Rückgabewert besitzen. Sie können wie normale Funktionen überladen werden. Wird kein Konstruktor deklariert, so wird vom Compiler der Default-Konstruktor (ohne Argumente) definiert, der einfach garnichts macht.

6.3 Destruktor

Der Destruktor (im Unterschied zu Javas `finalize`) heißt `~Klassenname` und besitzt ebenfalls keinen Rückgabetyp und auch keine Übergabeparameter. Prinzipiell sollten die Destruktoren potentiell ableitbarer Klassen (also so ziemlich alle) immer(!) „virtual“ deklariert (s. nächsten Abschnitt) werden!

6.4 Virtual-Funktionen

Funktionen von Basisklassen können redefiniert werden. Wird allerdings über eine Referenz auf den Basistyp eines Objekts die Funktion aufgerufen, so wird auch die Definition des Basisklasse verwendet. Anders, wenn die Funktion in der Basisklasse als „virtual“ deklariert wurde. In diesem Fall wird immer(!) die Redefinition der abgeleiteten Klasse verwendet. Die Virtual-Deklaration erfolgt durch das Voranstellen des Schlüsselwortes `virtual` vor die Funktions-Deklaration.
Ein Beispiel:

```
class Basisklasse {
  void a() const;
  virtual void b() const;
};

class Abgeleitet: public Basisklasse {
  void a() const;
  virtual void b() const;
};

void Basisklasse::a() const {
  cout << "a-Basis" << endl;
}
```

```
void Basisklasse::b() const {
  cout << "b-Basis" << endl;
}

void Abgeleitet::a() const {
  cout << "a-Abgeleitet" << endl;
}

void Abgeleitet::b() const {
  cout << "b-Abgeleitet" << endl;
}

main () {
  Abgeleitet abgeleitet;
  Basisklasse &basis = abgeleitet;

  abgeleitet.a();                // Ausgabe ‚a-Abgeleitet‘
  abgeleitet.b();                // Ausgabe ‚b-Abgeleitet‘
  basis.a();                     // Ausgabe ‚a-Basis‘
  basis.b();                     // Ausgabe ‚b-Abgeleitet‘
}
```

6.5 Klassen-Definition

Eine Klasse wird definiert über die Definition ihrer Komponenten (alle Funktionen und die statischen Variablen; die ‚einfachen‘ Variablen werden ja bei der Erzeugung eines Objekts der Klasse angelegt). Funktionen werden wie ‚normale‘ Funktionen auch definiert. Zu beachten ist dabei lediglich den ‚vollen‘ Funktionsnamen anzugeben (Sichtbarkeitsbereich+Funktionsname, z.B. „`void Abgeleitet::b()`“). Auch die statischen Variablen müssen einmal global deklariert werden, damit Speicherplatz für sie angelegt wird (auch hier den Variablennamen um den Sichtbarkeitsbereich erweitern).

6.6 `this`-Pointer

Innerhalb (nicht-statischer) Member-Funktions-Definitionen kann über die Pointer-Variable `this` auf das Objekt selbst zugegriffen werden (das Java-Konstrukt `super` existiert nicht, wäre bei Mehrfach-Vererbung auch nicht eindeutig definiert). `this` ist ein Pointer auf das Objekt selbst, also vom Typ `Objekttyp*`.

6.7 Konstruktor-Definition

Eine Besonderheit bei der Definition von Konstruktoren gegenüber der Definition einer einfachen Funktion ist die Angabe einer sog. „Initialisierungsliste“. In dieser wird festgelegt, welche Parameter dem Konstruktor der Basisklasse übergeben werden (anstelle des `super`-Aufrufs in Java). Ausserdem können in der Initialisierungsliste auch (nicht-statische) Klassenvariablen initialisiert werden. Bemerkung: Wird keine Init.-liste angegeben, so wird (genau wie in Java) der Konstruktor der Basisklasse ohne Parameter aufgerufen, falls er existiert. Ein Beispiel:

```
class Basisklasse
{
  Basisklasse(int av, int bv);
  virtual ~Basisklasse();

  int a;
  int b;
};

class Abgeleitet: public Basisklasse
{
  Abgeleitet(int bv, int cv);

  int c;
};

Basisklasse::Basisklasse(int av, int bv)
  : a(av), b(bv) // Initialisierungsliste für eigene Member-Variablen
{}

Basisklasse::~Basisklasse()
{}

Abgeleitet::Abgeleitet(int bv, int cv)
  : Basisklasse(0, bv), c(cv) // Init.-liste für Konstr. der Basisklasse
                              // und eigene Member-Variable c
{}
```

6.8 Referenzierung von Klassen-Komponenten

Nicht-statische Klassenkomponenten werden (wie in Java) über den „.“-Operator angesprochen. Besitzt man nur einen Zeiger auf das Objekt so kann man abkürzend für `(*Variablenname).Komponente` auch `Variablenname->Komponente` schreiben. Auf statische Komponenten kann man zugreifen, wie auf jede andere Funktion oder Variable auch. Man muß nur über den Scope-Operator eindeutig machen, auf welchen Namensbereich man sich bezieht, also: `Klassenname::Komponente` (s. auch den folgenden Abschnitt).

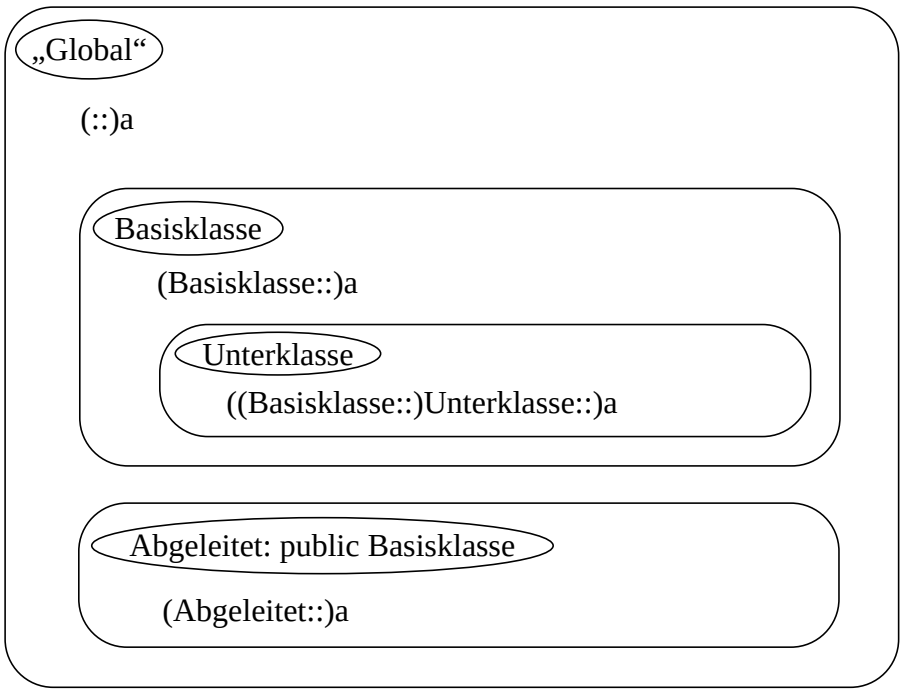
7 Sichtbarkeitsbereiche von Bezeichnern und Namensräume

Sichtbar sind immer alle diejenigen Bezeichner, die im selben Block (ein Block wird durch { und } begrenzt) oder umgebenden Blöcken bereits deklariert oder definiert wurden, und zwar bei Namensgleichheit immer die zuletzt eingeführten. Der äußerste Block ist der globale Bereich außerhalb aller geschweiften Klammern. Deklarationen und Definitionen, die hier gemacht werden, sind also überall bekannt.
Alle(!) Bezeichner die innerhalb eines Sichtbarkeitsbereichs eingeführt wurden, sind auch referenzierbar. Der Sichtbarkeitsbereich kann durch Klassen in Unternamensräume strukturiert werden. Namensräume schränken aber die Sichtbarkeit auf Bezeichner in anderen Namensräumen nicht(!) ein. Jede Klasse stellt

einen Namensraum (der nach dem Namen der Klasse benannt ist) bereit. Innerhalb eines Namensraums müssen Bezeichner eindeutig sein.
Befindet sich der Programmfluß innerhalb eines Namensraums (in einer Member-Funktion einer Klasse), so ist die Sichtbarkeit zwar auf die Bezeichner dieses Namensraumes fokussiert, jedoch nicht beschränkt. Über den „Scope“-Operator „::“ läßt sich auch auf die Bezeichner aller anderen Namensräume des aktuellen Sichtbarkeitsbereichs referenzieren. Das klingt etwas abstrakt. Ein Beispiel kann aber vielleicht etwas Klarheit bringen. Gegeben ist folgendes Programm:

```
void a() {
  cout << "Global" << endl;
}

class Basisklasse {
  void a() const;

  class Unterklasse {
    void a() const;
    void print() const;
  };
};

class Abgeleitet: public Basisklasse {
  void a() const;
  void print() const;
};

void Basisklasse::a() const {
  cout << "Basisklasse" << endl;
}

void Abgeleitet::a() const {
  cout << "Abgeleitet" << endl;
}

void Abgeleitet::print() const {
  a();                              // Ausgabe ‚Abgeleitet‘
  Basisklasse::a();                 // Ausgabe ‚Basisklasse‘
                                    // (auch wenn sie virtual wäre!)
  ::a();                            // Ausgabe ‚Global‘
}

void Basisklasse::Unterklasse::a() const {
  cout << "Unterklasse" << endl;
}

void Basisklasse::Unterklasse::print() const {
  a();                              // Ausgabe ‚Unterklasse‘
  ::a();                            // Ausgabe ‚Global‘
}
```

Die folgende Grafik versucht die Struktur, der eben definierten Klassen übersichtlich wiederzugeben:



Man kann sich das auch anders vorstellen. Es gibt nur den globalen Namensraum mit der Liste der folgenden Bezeichner:

```
a
Basisklasse::a
Basisklasse::Unterklasse::a
Abgeleitet::a
```

Befindet man sich in einem bestimmten Namensraum, so erweitert der Compiler (sofern der Namensraum nicht explizit angegeben ist) automatisch die referenzierten Bezeichner durch Voranstellen des aktuellen Namensraums. Befindet man sich also in der Funktion `Abgeleitet::print`, so wird aus dem Aufruf der Funktion `a` der Aufruf der Funktion `Abgeleitet::a`.

8 Umwandeln von Datentypen (Casting)

C++ erlaubt die Umwandlung von Datentypen in andere Datentypen. Dieses „Casten“ sollte zwar nach Möglichkeit vermieden werden, kann aber unumgänglich sein, wenn man z.B. verschiedene abgeleitete Klassen gleichermassen über einen Zeiger auf die Basisklasse behandeln will und danach zurück auf die eigentlich Klasse muss.
Dabei werden bestimmte Typumwandlungen automatisch vorgenommen, andere müssen explizit angegeben werden. Automatisch castet C++ beispielsweise `int` nach `long int` oder `int` nach `float`, also alle verlustfreien Konvertierungen. Automatisch werden auch Pointer und Referenzen von Objekten auf Pointer bzw. Referenzen einer Basisklasse der Klasse, von der das Objekt eine Instanz ist, gecastet (was ja

auch „verlustfrei“ bzw. konsistent geht, da die über die Basisklasse referenzierbaren Objektkomponenten ja alle bei der Ableitung geerbt wurden).
Jedes andere Casten bedarf der expliziten Angabe eines Casting-Operators. Hier gibt es eine alte Vorgehensweise, bei der wie in Java der Operator aus dem neuen Typ gebildet wird:

```
Basisklasse* a;
AbgeleiteteKlasse* b;
b = (AbgeleiteteKlasse*) a;
```

Dabei gibt es aber *keine* Überprüfung, ob das Casten möglich ist. D.h. es kann zu Abstürzen kommen, falls die beiden Klassen nicht ineinander überführbar sind!
Aus diesem Grund wurden eigene Cast-Operatoren eingeführt, die im Nicht-Erfolgsfalle einfach einen `NULL`-Pointer liefern. Diese sollten immer verwendet werden!

- `b = dynamic_cast<AbgeleiteteKlasse*>(a);`
  Erlaubt Casten eines polymorphen Typs (hier `Basisklasse*`) in den wahren Typ. Liefert NULL, falls nicht möglich und schmeisst zusätzlich eine Execpetion.
- `b = static_cast<Klasse2*>(a);`
  Konvertiert die Variable nur logisch. Kann man sich vorstellen, als die Erzeugung eines temporären Objekts des neuen Typs.

9 Ein- und Ausgabe mit Streams

Die Ein- und Ausgabe von Daten erfolgt in C++ über Streams (Datenströme). Ein Stream ist auch ein Objekt mit einer Reihe von Operatoren. Streamklassen sind von den Basisklassen `istream` (Daten einlesen) und `ostream` (Daten ausgeben) abgeleitet. Davon abgeleitet gibt es noch speziellere Streamklassen (z.B. `ifstream` und `ofstream` für das Lesen und Schreiben von Dateien).

9.1 Ausgabe

Für die Ausgabe stehen in C++ standardmäßig zwei Streams zur Verfügung, `cout` und `cerr`.
Dies sind beides Instanzen der Klasse `ostream`. `cout` ist die Standardausgabe und `cerr` die Fehlerausgabe (beide landen standarmäßig auf dem Bildschirm). Um in einen Stream zu schreiben gibt es den <<-Operator. Dieser Operator ist für alle fundamentalen C++-Datentypen überladen. Er liefert eine Referenz auf das Stream-Objekt selbst zurück, so daß er auch verkettet angewendet werden kann.
Beispiel:

```
cout << "Ausgabe: 5 + 6 = " << 5+6 << "." << endl;
```

`endl` erzeugt einen Zeilenvorschub.

9.2 Eingabe

Für die Eingabe stellt C++ ein Streamobjekt (Instanz von `istream`) namens `cin` zur Verfügung, welches standarmäßig von der Tastatur einliest. Auch `cin` ist für alle fundamentalen Datentypen von C++ überladen und kann folgendermaßen verwendet werden:

```
int i;
float f;
char string[80];
```

```
cin >> string >> i >> f;
```

10 Klassenbibliothek

C++ stellt in seinem aktuellen ANSI-Standard diverse Sprachelemente und eine Standard-Klassenbibliothek („STL“) zur Verfügung. Die STL bietet z.B. Containerklassen für beliebige Typen oder eine String-Klasse und erhöht die Komfortabilität und Sicherheit von C++ erheblich. Zudem wurden ein Reihe neue Sprachkonstrukte eingeführt (die Cast-Operatoren kennt ihr schon), die ebenfalls Mächtigkeit und Sicherheit von C++ verbessern. Wir werden hier nicht alle diese Features dokumentieren, sondern verweisen auf z.B. das Buch „*The C++ Standard Library*“, Nicolai M. Josuttis, Addison-Wesley.

Da dieser neue Standard noch nicht lange von dem auf dem Techfak-System verfügbaren C++-Compiler unterstützt wird, haben wir in der AG WBS seit einiger Zeit eine selbstentwickelte Klassenbibliothek („BasisObjects“), die verschiedene praktische Grundfunktionen zur Verfügung stellt. Die WBS-Agenten greifen z.B. auf diese Bibliothek zurück. Aus diesem Grund wollen wir hier einen kurzen Überblick über die BasicObjects geben. Es ist euch überlassen, ob ihr in euren Programmen die STL oder z.B. unsere Bibliothek (oder beides) benutzt.

Um die Basisfunktionalität der BasicObjects nutzen zu können, muß man lediglich eigene Klassen von der Basisklasse `BasicObject` ableiten. Funktionen, die man dann zur Verfügung gestellt bekommt, sind beispielsweise das typsichere Casten und die Verwaltung von typgemischten Containern (Listen, Tupel, ...).
Neben den fundamentalen C++-Datentypen, die als BasicObject-Datentypen implementiert wurden, gibt es u.a. eine relativ komfortable String-Klasse (`BO_String`), eine Listenklasse (`ObjectList`) und eine Tupelklasse (`NTupel`). Unter `/vol/wbs/include/BasicObject.hh` wird beschrieben, wie man eigene Klassen von BasicObject korrekt ableitet. Im selben Verzeichnis sind in `BO_BasicTypes.hh` die Beschreibungen der fundamentalen Datentypen (und der Stringklasse) und in `BO_Container.hh` die Listen- und die Tupelklasse beschrieben.

10.1 Compilieren mit BasicObject-Klassen

Compilieren:

```
CC <Dateiname> -o <Executablename> -I/vol/wbs/include -L/vol/wbs/lib
  -R/vol/wbs/lib -lBO
```

Objectfile erzeugen:

```
CC -c <Dateiname> -I/vol/wbs/include
```

Linken:

```
CC <Objectfile> <Objectfile> <...> ... -o <Executablename> -L/vol/wbs/lib
  -R/vol/wbs/lib -lBO
```

In unseren Makefiles werden die BasicObjects schon automatisch eingebunden.

10.2 BasicObject-Klassen

Im Headerfile:

```
#include <BasicObject.hh>
```

```
class BO_Klasse: public BasicObject {
  BO_FUNKTIONALITAET (BO_Klasse)

  [...]
};
```

Im Sourcefile:

```
BO_FUNKTIONALITAET (BO_Klasse, BasicObject)

[...]
```

Die wichtigsten Funktionen, die man erhält:

- Sicheres Casten:

```
BO_Klasse* boobj = new BO_Klasse ();
[...]
BasicObject* bo = boobj;
[...]
BO_Klasse* boobj2 = BO_Klasse::cast(bo);
BO_Klasse* boobj3 = BO_Klasse::strict_cast(bo);
```

- Verarbeitung in Containern (auch gemischt mit verschiedenen Objekttypen).
- Ein- und Ausgabe von Objekten über Stream-Operatoren (>>, <<) durch Implementation der Funktionen `drucken` und `einlesen`.

10.3 ObjectList und OLIterator

```
#include <BO_Container.hh>

ObjectList olist;
OLIterator it;
BOTyp* obj;

// Objekte in Liste einfügen
olist.addFirst(new BOTyp ());
olist.addLast(new BOTyp ());

// Liste mit Iterator auslesen
it = olist;
while(++it) {
  obj = BOTyp::strict_cast(it());
  [...]
}

// Suchen in der Liste
obj = BOTyp::cast(olist.first());
obj = BOTyp::cast(olist.last());
if (olist.contains(obj)) {[...]}
obj = BOTyp::cast(olist.containsValue(Value));
obj = BOTyp::cast(olist.containsValue(Value, Function));
```

```
// Löschen aus der Liste
olist.removeFirst();
olist.removeLast();
olist.remove(obj);
olist.clear();

// Testen der Liste
if (olist.isEmpty()) {[...]}
if (olist.notEmpty()) {[...]}
olist.size();
// Listenstatus ändern
olist.protectElements();          // Defaultmäßig werden Objekte, die aus
                                  // der Liste entfernt werden (z.B. durch
                                  // clear) gelöscht. Durch
                                  // protectElements wird dies verhindert.
                                  // Befindet sich ein Objekt in zwei
                                  // Listen, sollte mindestens eine davon
                                  // protected sein.
                                  // Listen können auch gleich protected
                                  // deklariert werden:
                                  // ObjectList olist (OC_PROTECT);
olist.unprotectElements();        // Gibt die Objekte wieder zum Löschen
                                  // frei.
```

Jedes BasicObject selbst kann auch protected werden (Default ist `unprotect`):

```
BOTyp obj;
obj.protect();
```

Es wird dann selbst dann nicht gelöscht, wenn die Liste, in der es sich befindet, unprotected ist.

10.4 NTupel

```
#include <BO_Container.hh>

// Tupel beliebiger Größe anlegen
NTupel nt (3);
BOTyp* obj;

// Beschreiben und Auslesen über Index-Operator
nt[1] = new BO_String („xyz“);
nt[2] = new BOTyp ();
nt[3] = new BOTyp ();
obj = BO_String::strict_cast(nt[2]);

// Tupel löschen
nt.clear();

// Testen des Tupels
```

```
nt.dim();

// Tupelstatus ändern
nt.protectElements();
nt.unprotectElements();
nt.protectMap(ocprotflag1, ocprotflag2);
[...]
nt.protectMap(ocprotflag1, ..., ocprotflag5);
nt.protectPlace(n);
nt.unprotectPlace(n);
// ocprotflag ::= OC_PROTECT | OC_UNPROTECT
// Listensuchfunktionen
NTupel *ntp;
ObjectList olist;
olist.add(&nt);
ntp = NTupel::cast(olist.containsValue(BO_String („xyz“),
                                       NTupel:: isKeyVal));
```

10.5 BO_String

```
#include <BO_BasicTypes.hh>

// Verschiedene Konstruktoren und Zuweisungsoperatoren
BO_String zahl (5);
zahl = 3.5;

BO_String string ("Text1");
string = "Text2";
string.clear ();

// Beschreibbar wie ein Stream
string << "5 + 6 = " << 5+6 << "." << STRING_END;

// Auslesbar wie ein Stream
BO_String word;
int num;
string = "Wert 35";
string >> word >> num;
while ((string >> word).ok()) {[...]}

// Liest auf verschiedene Arten aus Streams
string.instream_word();           // (default) oder...
string.instream_line();           // oder...
string.instream_buffer();
cin >> string;

// Kann in einen Stream geschrieben werden
cout << string;

// Zustandsabfrage
```

```
if (string.isEmpty()) {[...]}
if (string.notEmpty()) {[...]}
string.length();

// Vergleichsoperatoren
BO_String string1, string2;
if (string1 == string2) {[...]}
if (string1 == "xyz") {[...]}



// Test, an welcher Stelle Wort in Wortliste enthalten ist
switch (word.is_in("wort1 wort2 ...")) {
  case 1:                          // wort1
    [...]
    break;

  case 2:                          // wort2
    [...]
    break;

  default:
    [...]
    break;
}

// Und viele, viele weitere Funktionen...
```

11 Literatur

C++ für C-Programmierer
RRZN (Regionales Rechenzentrum für Niedersachsen / Universität Hannover)
< 10,- DM (zu beziehen beim Dispatcher im HRZ)

Objektorientiertes Programmieren in C++
Nicolai Josuttis
Addison-Wesley
79,90 DM

Design und Entwicklung von C++
Bjarne Stroustrup
Addison-Wesley
79,90 DM

The C++ Standard Library
Nicolai Josuttis
Addison-Wesley
49,95 US$

12 Beispiele für Übungsaufgaben

12.1 Hello world!

Spricht für sich. Wer‘s nicht kennt: Ein Programm, das einfach den Text „Hello world!“ auf dem Bildschirm ausgibt.

12.2 Graphische Objekte

Implementation eine Klasse ‚GraphischesObjekt‘, die als Basisklasse für beliebige graphische Objekte einer 2D-Graphik dienen können. Komponenten können sein: Position des Objekts, Lage der Eckpunkte, Farbe, etc. Als Methoden eignen sich Funktionen zum Setzen und Abfragen der Komponenten, aber auch eine Zeichnen-Funktion, die allerdings noch abstrakt ist, später aber die gezeichneten Kanten ausgibt (in Ermangelung eines graphischen Displays). Als Unterklasse könnte beispielsweise so etwas wie ein 2D-Eckpunkt sinnvoll sein.
Von ‚GraphischesObjekt‘ können verschiedene konkrete Objekte abgeleitet werden: Kreis, Dreieck, Rechteck, beliebiger Polygonzug, Text, etc., die die Eigenschaften von GraphischesObjekt erben, konkrete Eigenschaften hinzugewinnen (z.B. Radius oder Eckpunkte) und noch offene Funktionen implementieren (z.B. die Zeichnen-Funktion).

12.3 Graphische Szene

Implementation einer Klasse ‚GraphischeSzene‘, die eine Menge von graphischen Objekten der vorigen Aufgabe enthalten kann. Sie sollte zumindest eine Funktion ‚Zeichnen‘ enthalten, die alle graphischen Objekte der Szene darstellt. Weitere vorstellbare Komponenten könnten eine allgemeine Hintergrundfarbe oder eine allgemeine Skalierung der Objekte sein.

12.4 Agent

Implementation einer Klasse ‚Agent‘. Jeder Agent besitzt einen eindeutigen Namen und eine Funktion, mit der ein beliebiger String an einen beliebigen anderen Agenten geschickt werden kann. Selbstverständlich auch eine Funktion, mit der so ein String empfangen und verarbeitet werden kann. Sinnvoll als Klassenkomponente ist eine statische Liste, in der sich jeder Agent selbst einträgt (am besten gleich in seinem Konstruktor - `this`-Pointer!).
Von dieser allgemeinen Klasse ‚Agent‘ könnte man nun verschiedene speziellere Agenten ableiten, die beispielsweise eine besondere ‚String-empfangen-und-verarbeiten‘-Funktion besitzen.

12.5 Übung für BO-Klassen

Die folgende Aufgabe mal ohne die Definition eigener Klassen lösen, um mit den Listen, Tupeln, und anderen BasicObject-Typen vertraut zu werden.
Gegeben sei eine Lagerhalle als eine Liste. In dieser Lagerhalle sind verschiedene Produkte (z.B. unterschiedliche Lebensmittel) in unterschiedlicher Stückzahl gelagert. Jedes Produkt hat einen Namen, einen Preis und eine Liste von Zutaten. Es ergibt sich als folgende Struktur von Datentypen:

```
{(Stückzahl, (Produktname, Preis, {Zutat}))}
{(BO_Integer, (BO_String, BO_Float, {BO_String}))}
                                    // {} = ObjectList
                                    // () = NTupel
```

Es sei das Lager mit folgenden Produkten gegeben:

500 Tütensuppen zu 1,99 mit Tomatenersatzpulver, Aroma, Geschmacksverstärker
300 Zitronenkuchen zu 2,49 mit Mehl, Wasser, Milchpulver, Ei-Erzeugnisse, Aroma, Farbstoff
100 Obstwichtel zu 1,29 mit Wasser, Gelatine, Joghurt, Aroma, Farbstoff
150 Brause zu 2,69 mit Wasser, Kohlensäure, Aroma

Nun kann man beispielsweise folgende Funktionen implementieren:

Alle Produkte finden, die in einer bestimmten Stückzahl vorhanden sind.
Die Stückzahl eines bestimmten Produkts erhöhen oder vermindern.
Alle Produkte 1,- teurer machen.
Alle Produkte finden, die Aroma enthalten.