ProgCPP

J. Brupbacher

5. August 2019

Auf Vorlage der Zusammenfassung von L. Leuenberger, M. Ehrler, C. Ham, L. Däscher aus FS14 und der Vorlesung von R. Bonderer aus FS19. Die ganze Zusammenfassung richtet sich nach der `C++11` Version.

Inhaltsverzeichnis

1 Ein- und Ausgabe

Um die *C++* Ein- und Ausgaben nutzen zu können, muss man die Bibliothek iostream einbinden. Das geschieht mit:

```
#include <iostream>
```

Damit die Ein- und Ausgabebefehle auch wirklich genutzt werden können, müssen sie mithilfe von

```
using namespace std;
```

noch bekanntgegeben werden.

1.1 Streamkonzept

- Ein *Stream* repräsentiert einen sequentiellen Datenstrom.
- *C++* stellt 4 Standardstreams zur Verfügung:
  - *cin*: Standard-Eingabesteam
  - *cout*: Standard-Ausgabestream
  - *cerr*: Standard-Fehlerausgabestream
  - *clog*: mit *cerr* gekoppelt
- Alle diese Streams können auch mit einer Datei verbunden werden.
- Alle Schlüsselwörter müssen immer ganz links auf der Zeile stehen!

1.2 Eingabebeispiele

```
int zahl1;
int zahl2;
cout << "Zwei ganze Zahlen: ";
cin >> zahl1 >> zahl2;
```

1.3 Ausgabebeispiele

```
cout << "Dieser Text steht nun in der Kommandozeile!";
cout << "Dieser Text schliesst sich direkt an...";
```

```
int zahl1 = 5;
int zahl2 = 3;
cout << zahl1 + zahl2 << ", " << zahl1 * zahl2 << endl;

Ausgabe:
8, 15
```

1.4 Formatierte Ein- und Ausgabe: Manipulatoren ohne Parameter

ios, eine Basisklasse von *iostream*, stellt verschiedene Möglichkeiten (Format Flags) zur Verfügung, um die Ein- und Ausgabe zu beeinflussen.

1.4.1 boolalpha

bool-Werte werden textuell ausgegeben.

```
bool b = true;
cout << boolalpha << b << endl;
cout << noboolalpha << b << endl;
```

```
Ausgabe:
true
1
```

1.4.2 showbase

Zahlenbasis wird gezeigt.

```
int n = 20;
cout << hex << showbase << n <<
   endl;
```

```
Ausgabe:
0x14
```

1.4.3 showpoint

Dezimalpunkt wird immer ausgegeben.

```
double a = 30;
cout << showpoint << a << endl;
```

```
Ausgabe:
30.000
```

1.4.4 showpos

Vorzeichen wird bei allen Zahlen angezeigt.

```
int p = 1, z = 0, n = -1;
cout << showpos << p << '\t' << z
     << '\t' << n << endl;
```

```
Ausgabe:
+1   +0
   -1
```

1.4.5 uppercase

Alles in Grossbuchstaben ausgeben.

```
cout << showbase << hex;
cout << uppercase << 77 << endl;
```

```
Ausgabe:
0X4D
```

1.4.6 dec, hex, oct

Ausgabe erfolgt in dezimal, hexadezimal oder oktal.

```
int n = 70;
cout << dec << n << endl;
cout << hex << n << endl;
cout << oct << n << endl;
```

```
Ausgabe:
70
46
106
```

1.5 Formatierte Ein- und Ausgabe: Manipulatoren mit Parameter

Hier muss zwingend *<iomanip>* eingebunden werden!

1.5.1 setw()

Angabe der Feldbreite.

```
cout << setw(8);
cout << 77 << endl;
```

```
Ausgabe:
      77
```

1.5.2 setfill()

Auffüllen mit beliebigem Füllzeichen.

```
cout << setfill('x') << setw (8);
cout << 77 << endl;
```

```
Ausgabe:
xxxxxx77
```

1.5.3 setprecision()

Angabe der Genauigkeit einer Zahl.

```
double f =3.14159;
cout << setprecision(5) << f
     << endl;
cout << setprecision(7) << f
     << endl;
cout << fixed;
cout << setprecision(5) << f
     << endl;
cout << setprecision(7) << f
     << endl;
```

```
Ausgabe:
3.1416
3.14159
3.14159
3.1415900
```

1.6 Streams und Dateien

Streams sind Abstraktionen für die zeichenweise Ein-/Ausgabe. Dabei ist es egal, ob die Ausgabe auf den Bildschirm erfolgt oder auf eine Datei. Dateien können also über die normalen Möglichkeiten der Ein-/Ausgabe geschrieben bzw. gelesen werden.

1.6.1 Öffnen von Dateien

Das Öffnen einer Datei erfolgt beim Anlegen eines Stream-Objektes beziehungsweise über die Funktion *open*. Dabei werden der Name der Datei und der Öffnungsmodus als Parameter übergeben.

```
// Default-Modi
ifstream readfrom("Eingabe.txt");
ofstream writeTo("Ausgabe.txt");

// Für Eingabe öffnen
ifstream readFrom("Eingabe.txt", ios_base::in);

// Binäre Eingabe
ifstream readFrom("Eingabe.txt", ios_base::in
                               | ios_base::bin);

// Binäre Ausgabe und Datei leeren
// falls sie existiert
ofstream writeTo("Ausgabe.txt", ios_base::out
                              | ios_base::bin
                              | ios_base::trunc);
```

Modus	Kommentar
in	Datei für Eingabe öffnen
out	Datei für Ausgabe öffnen
app	Schreiboperationen am Dateiende ausführen
ate	Nach dem Öffnen der Datei sofort an das Dateiende verzweigen
trunc	Zu öffnende Datei zerstören, falls sie bereits existiert
bin[ary]	Ein-/Ausgabe wird binär durchgeführt und nicht im Textmodus

1.6.2 Lesen und Setzen von Positionen

Die aktuelle Position innerhalb einer Datei kann beliebig verändert werden. Dazu stehen folgende Typen und Methoden zur Verfügung:

- *streampos* ist der Typ einer Dateiposition.
- *seekg(offset, direction)* zum Beispiel setzt die aktuelle Dateiposition einer Datei. *offset* gibt die Position vom Dateianfang bzw. -ende aus an, *direction* legt fest, von wo aus die Position bestimmt wird.
  - *ios::beg*: Dateianfang
  - *ios::cur*: Aktuelle Position
  - *ios::end*: Dateiende
- *tellg* liefert die aktuelle Position in der Datei.
- *seekp* und *tellp* sind die entsprechenden Versionen für die Put-Varianten.

2 Ausdrücke und Operatoren

Ähnlich wie mathematische Ausdrücke stellen auch Ausdrücke in *C++* Berechnungen dar und bestehen aus Operanden und Operatoren. Die Auswertung jedes Ausdrucks liefert einen Wert, der sich aus der Verknüpfung von Operanden durch Operatoren ergibt.

2.1 Auswertungsreihenfolge

Priorität	Operator	Beschreibung	Assoziativität
1	`::`	Bereichsauflösung	von links nach rechts

2	`++ --`
	`()`
	`[]`
	`.`
	`->`
		Suffix-/Postfix-Inkrement und -Dekrement
		Funktionsaufruf
		Arrayindizierung
		Elementselektion einer Referenz
		Elementselektion eines Zeigers
			von links nach rechts

3	`++ --`
	`+ -`
	`! ~`
	`(type)`
	`*`
	`&`
	`sizeof`
	`new, new[]`
	`delete, delete[]`
		Präfix-Inkrement und -Dekrement
		unäres plus und unäres Minus
		logisches NOT und bitweises NOT
		Typkonvertierung
		Dereferenzierung
		Adresse von
		Typ-/Objektgrösse
		Reservierung Dynamischen Speichers
		Freigabe Dynamischen Speichers
			von rechts nach links

4	`.* ->*`	Zeiger-auf-Element	von links nach rechts
5	`* / %`	Multiplikation, Division und Rest	von links nach rechts
6	`+ -`	Addition und Subtraktion	von links nach rechts
7	`<< >>`	bitweise Rechts- und Linksverschiebung	von links nach rechts
8	`< <=` `> >=`	kleiner-als und kleiner-gleich grösser-als und grösser-gleich	von links nach rechts
9	`== !=`	gleich und ungleich	von links nach rechts
10	`&`	bitweises AND	von links nach rechts
11	`^`	bitweise XOR	von links nach rechts
12	`|`	bitweises OR	von links nach rechts
13	`&&`	logisches AND	von links nach rechts
14	`||`	logisches OR	von links nach rechts

15	`?:`
	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		bedingte Zuweisung
		einfache Zuweisung
		Zuweisung nach Addition/Subtraktion
		Zuweisung nach Multiplikation, Division, Rest
		Zuweisung nach Links-, Rechtsverschiebung
		Zuweisung nach bitweisem AND, XOR und OR
			von rechts nach links

16	`throw`	Ausnahme werfen	von rechts nach links
17	`,`	Komma (Sequenzoperator)	von links nach rechts

(Priorität 1 hat Vorrang vor allen anderen)

2.1.1 Assoziativität

Die Assoziativität gibt Auskunft über die Auswertungsreihenfolge der Operanden eines Ausdrucks. So wird zum Beispiel im Ausdruck *p++* zuerst *p* ausgewertet und dann die linke Seite des Operators *++* (*p*) erhöht, während der Ausdruck *++p* zuerst *p* erhöht und dann den Ausdruck auswertet.

```
int i = 6;
cout << i++;  // gibt i (= 6) aus und erhoeht danach i
cout << ++i;  // erhoeht i (= 7+1) und gibt i dann aus
```

2.1.2 Priorität

Die Priorität von Operatoren wiederum gibt an, in welcher Reihenfolge die verschiedenen Operanden eines Ausdrucks ausgewertet werden. Die multiplikativen Operatoren weisen zum Beispiel eine höhere Priorität als die additiven Operatoren auf.

3 Funktionen

Im Vergleich zu *C* funktionieren Funktionen in *C++* genau gleich, zusätzlich gibt es aber einige neue, nützliche Eigenschaften:

- Vorbelegung von Parametern (Default-Argumente)
- überladen von Fuktionen (Overloading)
- Operatorfunktionen
- inline-Funktionen

3.1 Überladene Funktionen

- Die Identifikation einer Funktion erfolgt über die Signatur, nicht nur über den Namen. Die Signatur besteht aus dem Namen der Funktion plus der Parameterliste (Reihenfolge, Anzahl, Typ). Der Returntyp wird nicht berücksichtigt.
- Der Name der Funktionen ist identisch.
- Die Implementation muss für jede überladene Funktion separat erfolgen.
- Overloading sollte zurückhaltend eingesetzt werden. Wenn möglich sind Default-Argumente vorzuziehen.

```
void print(char c);
void print(int i);
void print(double d);
```

3.2 Vorbelegte Parameter

- Parametern können im Funktionsprototypen Defaultwerte zugewiesen werden.
- Beim Funktionsaufruf können (aber müssen nicht) die Parameter mit Defaultwerten weggelassen werden.
- Achtung: Hinter (rechts von) einem Default-Argument darf kein nicht vorbelegter Parameter mehr folgen!

```
void prtDate(int day=1, int month=3, int year
    =2009);

// erlaubt sind z.B. die folgenden Aufrufe:
prtDate();                  // 1-3-2009
prtDate(23);                // 23-3-2009
prtDate(15, 6);             // 15-6-2009
prtDate(24, 7, 2012);       // 24-7-2012
```

3.3 *inline*-Funktionen vs. *C*-Makros

3.3.1 *C*-Makros

- *C*-Makros werden definiert mit *#define*.
- *C*-Makros bewirken eine reine Textersetzung ohne jegliche Typenprüfung.
- Bei Nebeneffekten (welche zwar vermieden werden sollten) verhalten sich Makros nicht wie beabsichtigt.
- *C*-Makros lösen zwar das Problem mit dem Overhead, sind aber sehr unsicher.

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

3.3.2 *inline*-Funktionen

- Lösen das Overhead-Problem.
- Textersetzung mit Typenprüfung.
- Einsetzen wenn der Codeumfang der Funktion sehr klein ist und die Funktion häufig aufgerufen wird (z.B. in Schleifen).
- Rekursive Funktionen und Funktionen, auf die mit einem Funktionspointer gezeigt wird, werden nicht ge-*inlined*.

```
inline int max(int a, int b)
{
  return a > b ? a : b;
}
```

4 Höhere und strukturierte Datentypen

4.1 Pointer

Pointer sind in *C++* zu 99% wie in *C*.

4.1.1 Null-Pointer

Ab *C++11* gibt es einen vordefinierten Wert für den Nullpointer: *nullptr*

```
int* ptr = nullptr;
```

4.1.2 *void*-Pointer

- Wenn bei der Definition des Pointers der Typ der Variable, auf die der Pointer zeigen soll, noch nicht feststeht, wird ein Pointer auf den Typ *void* vereinbart.
- Ein Pointer auf *void* umgeht die Typenprüfung des Compilers. Er kann ohne *typecast* einem typisierten Pointer zugewiesen werden aber er kann keine Zuweisung von einem typisierten Pointer erhalten (in *C* erlaubt).
- Jeder Pointer kann durch Zuweisung in den Typ *void** und zurück umgewandelt werden, ohne dass Informationen verloren gehen.

4.1.3 Pointer auf Funktionen

```
#include <stdio.h>
int foo(char ch)
{
  for (int i = 1; i <= 10; i++)
    printf("%c ", ch);
  return i;
}
int main(void)
{
  int (*p)(char);
    // Deklaration des Funktionspointers
  int ret;
  p = foo;
    // ermittle Adresse der Funktion foo()
  ret = p('A');
    // Aufruf von foo() ueber Funktionspointer
  return 0;
}
```

Vereinbarung eines Pointers

```
int (*p)(char);
```

ptr ist hier ein pointer auf eine Funktion mit Rückgabewert vom Typ *int* und einem Übergabeparameter vom Typ *char*. Die Klammern müssen unbedingt gesetzt werden.

Zuweisung einer Funktion

```
p = funktionsname;
  \\ oder
p = &funktionsname;
```

Aufruf einer Funktion

```
a = (*p) (Uebergabeparameter);
  \\ oder
a = p (Uebergabeparameter);
```

4.1.4 const bei Pointern

4.1.4.1 konstanter String

```
const char* text = str;
```

4.1.4.2 konstanter Pointer

```
char* const text = str;
```

4.1.4.3 konst. Pointer auf konst. String

```
const char* const text = str;
```

4.2 Referenzen

- Referenzen sind alternative Namen oder Alias für ein Objekt
- Beim Pointer möchte man eine Adresse, eine Referenz kann auch auf Register verweisen
- Wenn immer möglich sollten Referenzen verwendet werden, weil sicherer

```
int x = 24;
int& r1 = x; // Definition der Referenz r1
x = 55; // x == 55, r1 == 55 (dasselbe Objekt)
r1 = 7; // x == 7, r1 == 7 (dasselbe Objekt)
r1++; // x == 8, r1 == 8 (dasselbe Objekt)
```

4.3 Pointer und Referenzen als Rückgabewert und Parameterübergabe

Bei Variablenübergabe (call by value) werden Kopien übergeben, welche nicht verändert werden können.
Bei Referenzübergabe (call by reference) kann die Subroutine die Werte bleibend verändern.
Objekte einer Klasse und Strukturvariablen sollen immer by reference übergeben werden!

4.3.1 call by reference

```
void swap(int& a, int& b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
int main()
{
  int x = 4;
  int y = 3;
  swap(x, y); // OK!
  return 0;
}
```

```
void swap(int* a, int* b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
int main()
{
  int x = 4;
  int y = 3;
  swap(&x, &y); // OK!
  return 0;
}
```

4.3.2 call by value

```
void swap(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
int main()
{
  int x = 4;
  int y = 3;
  swap(x, y); // keine Ausw.
  return 0;
}
```

4.4 Dynamische Speicherverwaltung

4.4.1 pro Memoria: Variablen

- erleichtern u.a. den Zugriff auf Speicherstellen (anstelle Adressen)
- Müssen zur Entwicklungszeit im Code definiert werden
- Der Speicher einer Variable wird automatisch freigegeben, sobald die Variable nicht mehr gültig ist.

4.4.2 Dynamische Speicherverwaltung

- Speicher kann zur Laufzeit (dynamisch) vom System angefordert (alloziert) werden
  - Operator: *new* (in *C*: Funktion *malloc*())
- Dynamisch allozierter Speicher muss wieder explizit freigegeben werden
  - Operator: *delete* (in *C*: Funktion *free*())
- Dynamischer Speicher wird nicht auf dem Stack angelegt, sondern auf dem **Heap**
- Auf Dynamisch allozierter Speicher kann **nur** über Pointer zugegriffen werden

4.4.3 Syntax

```
int* pInt = new int;
  // Speicher fuer int alloziert
char* pCh1 = new char;
  // Speicher fuer char alloziert
char* pCh2 = new char;
  // Speicher fuer char alloziert
*pInt = 23;
std::cin >> *pCh1;
pCh2 = pCh1;
// pCh2 zeigt nun auch auf die gleiche
   Speicherstelle wie pCh1. Damit geht
   aber der Zugriff auf die Speicherstelle
    verloren, auf die pCh2 gezeigt hat (
   memory leak!)
delete pInt; // Speicher wieder freigeben
delete pCh1;
delete pCh2; // ergibt Fehler, bereits
   ueber pCh1 freigegeben
```

4.4.4 Vorsichtsmassnahmen

- Beim Anwenden des delete-Operator auf einen bereits freigegebenen Speicherbereich, kann Probleme verursachen.
- Oft wird deshalb ein Pointer nach der *delete*-Operation auf 0 (bzw. *nullptr*) gesetzt.

4.4.5 Memory Leak, Garbage Collection

- **Memory Leak** = Speicher, der nicht freigegeben wurde oder auf welchen der Zugriff verloren ging, belegt aber weiterhin Platz im Speicher. (wie ein Leck)
- In einigen Programmiersprachen (wie z.B. Java) gibt es einen sogenannten **Garbage Collector**, welcher dieser Speicher automatisch freigibt.
- In C++ gibt es keinen Garbage Collector. Der Programmierer ist selber dafür verantwortlich.

4.4.6 Dynamische Allozierung von Arrays

- mit `new[]` kann dynamischer Speicher für ein Array alloziert werden.
- Zugriff erfolgt wie bei statischen Arrays.
- Mit `delete[]` können dynamisch allozierte Arrays wieder explizit freigegeben werden.

```
int* pInt = new int[100];
pInt[22] = -45;
delete pInt; // Fehler: nur pInt[0] wird freigegeben
delete[] pInt; // korrekter Befehl
```

4.4.7 Dynamische Allozierung von Matrizen

4.4.7.1 Variante 1

- m*n-Matrix als eindimensionaler Array der Grösse (m*n)
- Zugriff nur über Pointer:

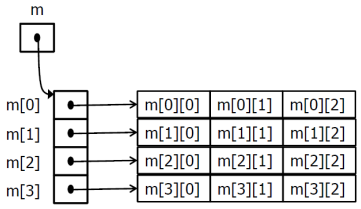
```
*((m+2)*(n+3)) = 23.44;
```

4.4.7.2 Variante 2

- Ein Array der Grösse m und dem Typ Array
- Auf ein Matrixelement kann somit über Arrayindizes zugegriffen werden:

```
matrix[2][3] = 23.44;
```

```
double** m = nullptr;
m = new double*[4];
for(int i=0 ; i<4 ; ++i)
  m[i] = new double[3];
m[2][1] = 34.675;
```



- Jedes *m[i]* ist ein Pointer auf ein Array mit 3 Elementen vom Typ *double*.
- *m[i]* selbst ist vom Typ *double**.
- der Speicher von Matrizen wird wie folgt freigegeben (zuerst jede Zeile dann Array mit *double**):

```
for(int i=0 ; i<4 ; ++i)
  delete[] m[i];
delete[] m;
```

4.4.7.3 Vor- und Nachteile Variante 2

- Nur mit der Variante 2 kann über Arrayindizes zugegriffen werden
- Die einzelnen Zeilen liegen mögicherweise nicht auf aufeinanderfolgenden Speicherstellen
- Der Zugriff ist effizienter, obwohl man zusätzliche Pointer benötigt

4.5 Structs

- Grundsätlich wie in *C*
- *typedef* braucht es nicht

Beispiel in *C*:

```
typedef struct
{
  int x;
  int y;
} Point;
```

Beispiel in *CPP*:

```
struct Point
{
  double x;
  double y;
};

Point p1;
```

5 Gültigkeitsbereiche, Namensräume und Sichtbarkeit

5.1 Namensräume

```
namespace MyLib1 {
  int i;
  void foo();
}

namespace MyLib2 {
  int i;
  void foo();
}

MyLib1::foo();
  // foo aus MyLib1
MyLib2::i = 17;
  // i aus MyLib2
```

Namensräume sind Gültigkeitsbereiche, in denen beliebige Bezeichner (Variablen, Klassen, Funktionen, andere Namensräume, Typen, etc.) deklariert werden können.

- Ein Namensraum kann deklariert werden. Alle enthaltenen Objekte werden diesem Namensraum zugeordnet. Auf Bezeichner eines Namensraumes kann mit dem Scope Operator *::* zugegriffen werden.
- Einem Namensraum kann ein so genannter Alias zugeordnet werden, über den er angesprochen wird.
  namespace FBSSLIB = Financial_Branch_and_System_Service_Library;
- Eine so genannte *Using*-Deklaration erlaubt den direkten Zugriff auf einen Bezeichner eines Namensraumes.
  using MyLib1::foo;
  foo();
- Mit einer so genannten *Using*-Direktive kann auf alle Bezeichner eines Namensraums direkt zugegriffen werden.
  using namespace MyLib1;
  foo();

5.1.1 namenlose Namensräume

Anstelle von *static* in *C*

```
namespace
{
  ...
}
```

5.2 Deklarationen

5.2.1 Speicherklassenattribute

- *auto*: gilt als Standard wenn nichts anderes steht. Gültigkeitsbereich der *auto* Variablen ist innerhalb des Blockes in dem sie deklariert wurde.
- *register*: Hinweis an den Compiler möglichst die Variable in einem Register abzulegen.
- *static*: Variablen leben von ihrer Deklaration bis zum Programmende. Geeignet um zum Bsp. Funktionsaufrufe zu zählen anstatt mit globaler Variable.
- *extern*: Zugriff auf eine *static* Variable in einem anderen File, welches zu einem gesamten Programm gelinkt wurde.
- *mutable*: Klassenelemente mit *const* oder *static* Attributen können nachträglich verändert werden.

5.2.2 Typqualifikatoren

- *const*: Objekte dürfen nicht verändert werden. RValues.
- *volatile*: Objekte werden evtl. von Aussen im Programmverlauf verändert, und dürfen daher vom Compiler nicht zu Optimierungszwecken zwischengespeichert werden. Sie werden immer aus dem Hauptspeicher eingelesen. Verlangsamt das Programm, sollte daher gezielt eingesetzt werden.

5.2.3 typedef

```
typedef int Number;
typedef int Vector[25]

Number a; //Deklaration einer int-Zahl
Vector s; //Deklaration eines 25er int-arrays
```

Das Schlüsselwort *typedef* ermöglicht die Einführung neuer Bezeichner, die dann im Programm anstelle von anderen Typen verwendet werden können. *typedef* führt allerdings keine neuen Typen, sondern Synonyme für einen existierenden Datentyp ein. Es ist also mehr oder weniger eine Textersetzung.

5.3 Type-cast

5.3.1 implizite-Typumwandlung

Ausdrücke werden bei einer Zuweisung automatisch in den erwarteten Typ umgewandelt.

```
cout << a; // erfordert RValue, falls a LValue wird es autom. konvertiert
float f = 1.23
int i = f; // Gleitkomma -> Integer (aufrunden, abrunden implementationsabhaengig)
f = 1;     // Integer -> Gleitkomma
```

5.3.2 *C*-Stil

```
int a = (int)4.6;
```

5.3.3 Funktions-Stil

```
int a = int(4.6);
```

Achtung: Den *C*- und Funktions-Stil sollte in *C++* nicht verwendet werden, da daraus der Grund für die Typumwandlung nicht erkannt wird und der Compiler keine Prüfung durchführt. Deshalb stellt *C++* die folgenden spezifischen und sichereren Typumwandlungen zur Verfügung.

5.3.4 Neu in $C++$:

5.3.4.1 *const_cast*

Ausschliesslich bei der Entfernung des *const*-Qualifikators.
Syntax:

```
const_char<Zieltyp>expression
```

Beispiel:

```
const char* findSubString(const char* str, const char* subStr)
{
  return strstr(const_cast<char*>str, const_cast<char*>subStr);
}
```

5.3.4.2 *static_cast*

Umwandeln eines Klassenobjekt in ein Objekt seiner Basisklasse. Syntax ist analog zu *const_cast*.
Beispiel:

```
static_cast<SuperHero>Batman;
```

5.3.4.3 *dynamic_cast*

Umwandlung von Polymorphen Objekten im Zusammenhang mit dem Typsystem von C++.
Beispiel:

```
dynamic_cast<SuperHero*>p;
```

5.3.4.4 *reinterpret_cast*

Neue Interpretation der zugrunde liegenden Bitkette.
Beispiel:

```
reinterpret_cast<int*>str // char Pointer str wird in int-Pointer gewandelt.
```

6 Module und Datenkapseln

Modul = Unit Modultest = Unittest

6.1 Motivation

- **Arbeitsteilung:** Grosse Programme werden von mehreren Personen entwickelt.
- **Effizienz:** Eine Übersetzungseinheit (Datei) muss bei jeder Änderung neu übersetzt werden (je grösser die Datei desto langsamer die Übersetzung)
- **Strukturierung:** Ein grosses Programm in mehrere vernünftige Teile (Baugruppen, Units) aufteilen (Divide and conquer)

6.2 Ziele der Modularisierung

- Klare, möglichst schlanke Schnittstellen definieren
- Units so bilden, das Zusammengehörendes in einer Unit isoliert wird (Kohäsion soll hoch sein)
- Schnittstellen zwischen den Units sollen klein sein (Kopplung soll klein sein)
- Abhängigkeiten unter den Units sollen eine Hierarchie bilden, zirkuläre (gegenseitige) Abhängigkeiten müssen vermieden werden

6.3 Vom Modul zur Datenkapsel

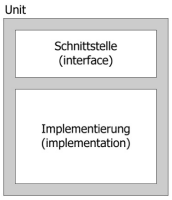
Eigenschaften einer Unit (eines Moduls):

- realisiert eine in sich abgeschlossene Aufgabe
- kommuniziert über ihre Schnittstelle mit der Umgebung
- kann ohne Kenntnisse ihres inneren Verhaltens in ein Gesamtsystem integriert werden (include Header)
- ihre Korrektheit kann ohne Kenntnis ihrer Einbettung in einem Gesamtsystem nachgewiesen werden (mittels Unittest)
- **Die Datenkapsel fordert nun zusätzlich, dass auf die Daten nicht direkt zugegriffen werden darf, sondern nur über Zugriffsfunktionen.**

Die Schnittstelle beschreibt, was das Modul zur Verfügung stellt, verbirgt dabei wie das Verhalten konkret realisiert ist (Geheimnisprinzip, Information Hiding). Der User der Unit darf keine Annahme über den inneren Aufbau machen. Der Entwickler der Unit kann deren inneren Aufbau verändern, solange die Schnittstelle dadurch nicht ändert.

6.4 Unitkonzept / Module und Datenkapseln in C++

- Interface definiert die Schnittstelle, d.h. die Deklarationen wie Funktionsprototypen, etc. (Schaufenster)
- Implementation: in diesem Teil sind die Unterprogramme definiert, d.h. auscodiert (Werkstatt)
- Das Interface wird in einer Headerdatei (**.h*) beschrieben, die Implementation liegt in einer **.cpp*- Datei



6.5 Die Schnittstellendatei

Jede *.h*-Datei enthält als erste Anweisungsfolge einen Include-Guard welcher Mehrfacheinfügen verhindert. Der Syntax lautet:

```
#ifndef FOO_H_
#define FOO_H_
// Deklarationen (Punkt 2-7 nachfolgende Liste)
#endif /* FOO_H_ */
```

Deklarationsreihenfolge in Headerdatei (*.h)

1. Dateikommentar
2. *#include* der verwendeten System-Header (*iostream*, etc.)
   #include <...>
3. *#include* der projektbezogenen Header (*#include "..."*)
4. Konstantendefinitionen
5. *typedef*s und Definition von Strukturen
6. Allenfalls extern-Deklaration von globalen Variablen
7. Funktionsprototypen, inkl. Kommentare der Schnittstelle, bzw. Klassendeklarationen

- **Achtung:** kein *using namespace* in Headerdateien!

6.6 Die Implementierungsdatei

Deklarationsreihenfolge in Implementierungsdatei (**.cpp*)

1. Dateikommentar
2. *#include* der verwendeten System-Header (*iostream*, etc.) *#include <...>*
3. *#include* der projektbezogenen Header (*#include "..."*)
4. Verwendung von *using namespace*
5. allenfalls globale Variablen und statische Variablen
6. Präprozessor-Direktiven
7. Funktionsprototypen von lokalen, internen Funktionen
8. Definition von Funktionen und Klassen (Kommentare aus Headerdatei nicht wiederholen!)

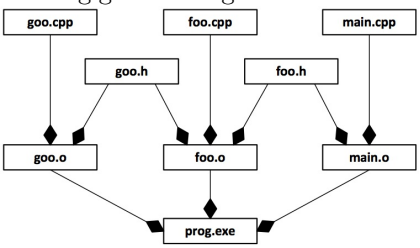
6.7 Buildprozess / Makefile

Der Buildprozess erstellt aus den einzelnen Dateien einen ausführbaren Code. Dazu werden zuerst alle **.cpp*-Files compiliert. Die daraus entstandenen Objektdateien müssen anschliessend gelinkt und somit zu einer ausführbaren Datei zusammengesetzt. Die Eingabe in der Konsole sieht wie folgt aus:

```
g++ -c foo.cpp  // compile Unit, do with all *.cpp
g++ -o foo.exe foo.o goo.o hoo.o  // link Unit
```

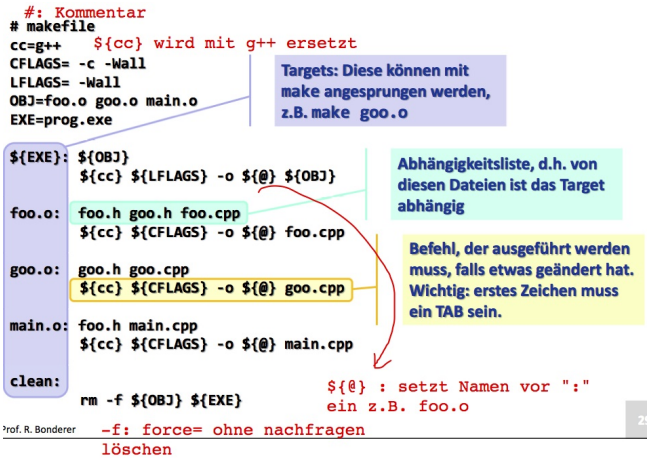
Es wäre mühsam, wenn diese Befehle jedesmal neu eingetippt werden müssten. Deshalb wird in der Praxis oft ein Buildtool eingesetzt, z.B. *make*.

Abhängigkeitsliste gemäss UML-Notation:



6.7.1 Make-File

- In einem *make*-File können Abhängigkeiten definiert werden
- Wenn eine Datei geändert wurde, dann werden alle Operationen ausgeführt mit den Dateien, welche von dieser geänderten Datei abhängen
- Der Befehl (*g++*) wird z.B. nur dann ausgeführt, wenn sich an den Dateien, zu denen eine Abhängigkeit besteht, etwas geändert hat



7 Klassenkonzept

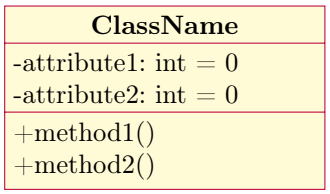
7.1 Begriff der Klasse

- Eine Klasse ist eine Struktur (eine Struktur besteht nur aus Daten), die mit den Funktionen, welche auf diesen Daten arbeiten, erweitert wurde.
- Eine Klasse ist also eine Struktur, welche die Daten und die Funktionen auf diesen Daten in ein syntaktisches Konstrukt packt.
- Die Klasse ist die Umsetzung der Datenkapsel.
- Eine Klassendeklaration ist eine Typendefinition. Die Variablen einer Klasse werden als Objekte bezeichnet.

7.2 UML ist...

- ... die Abkürzung für **U**nified **M**odelling **L**anguage
- ... eine graphische Modellierungssprache
- ... ein fortlaufendes (objektorientiertes) Modellierungskonzept für alle Software-Entwicklungsphasen (Ziel der UML)
- ... der Standard für Softwaremodellierung
- ... (programmier-)sprachunabhängig
- ... **kein** Softwareprozess-Modell
- ... **kein** Lebenszyklusmodell
- ... **keine** Programmiersprache
- ... **nicht** ohne Redundanz (es gibt oft mehrere Möglichkeiten, etwas zu modellieren)
- ... **kein** Softwaretool

7.3 UML-Notation einer Klasse



- Eine Klasse ist der Bauplan für Objekte.
- Eine Klasse besteht aus Daten (Attribute) und den Funktionen (Methoden) auf diesen Daten.
- Sichtbarkeit:
  - `+` : *public*
  - `-` : *private*
  - `#` : *protected*

7.4 Üblicher Aufbau einer Klassensyntax

```
class Classname // Deklaration der
    Klasse
{
  public:
    ...
  protected:
    ...
  private:
    ...
};  // Semikolon nicht vergessen
```

7.4.1 Zugriffsschutz

- *public* - Elemente können innerhalb und von ausserhalb der Klasse angesprochen werden.
  - fast alle Methoden sind *public*
  - Attribute sollen nie *public* sein
- *protected* - Elemente können von innerhalb der Klasse und von abgeleiteten Klassen angesprochen werden.
  - nur sparsam einsetzen!
- *private* - Elemente können nur innerhalb der Klasse angesprochen werden.
  - grundsätzlich für alle Attribute und für einzelne (lokale) Methoden

7.4.2 Operationen einer Klasse

Operationen eine Klasse (= Funktionen, die im Klassenrumpf definiert sind) werden als Elementfunktionen oder Methoden bezeichnet. Üblicherweise beginnen Elementfunktionen mit einem Kleinbuchstaben und werden in camelCase (mixedCase) notiert. *isEmpty();*

7.4.3 Information Hiding

- Klassen exportieren generell ausschliesslich Methoden. Alle Daten sind im Innern (private-Abschnitt) verborgen, der Zugriff erfolgt über die so genannten Elementfunktionen.
- Jede Klasse besteht damit aus zwei Dateien, der Schnittstellendatei (*.h*) und der Implementierungsdatei (*.cpp*).

7.4.3.1 *friend*-Elemente

- *friend* - Jede Klasse kann andere Klassen oder Funktionen zum Freund erklären. Dadurch werden die Zugriffsregeln durchbrochen.
- Jeder *friend* darf auf alle Elemente der Klasse zugreifen.
- *friend* ist eine *C++* - Spezialität, welche die meisten anderen Programmiersprachen (z.B. *Java*) nicht anbieten.
- *friends*, insbesondere *friend*-Klassen, können ein Anzeichen für schlechtes Design sein. Sie durchbrechen wichtige Prinzipien der objektorientierten Programmierung.

7.4.4 Beispiel an der Klasse Rechteck

```
// Klassendeklaration in rectangle.h
class Rectangle
{
  public:
    void setA(double newA);
    void setB(double newB);
    double getA() const;
    double getB() const;
    double getArea() const;
  private:
    double a;
    double b;
};
```

```
// Klassendefinition in rectangle.cpp
#include "rectangle.h"
void Rectangle::setA(double newA)
{
  a = newA;
}
void Rectangle::setB(double newB)
{
  b = newB;
}
double Rectangle::getA() const
{
  return a;
}
double Rectangle::getB() const
{
  return b;
}
double Rectangle::getArea() const
{
  return a*b;
}
```

Rectangle
-a : double -b : double
+setA(in newA : double) +setB(in newB : double) +getA() : double +getB() : double +getArea() : double

7.5 Elementfunktionen

- sind Funktionen, die in der Schnittstelle der Klasse spezifiziert sind.
- Elementfunktionen haben vollen Zugriff auf alle Klassenelemente (auch auf solche, die mit *`private:`* gekennzeichnet sind.
- Auf Elementfunktionen kann nur unter Bezugnahme auf ein Objekt der Klasse, bzw. mit dem Scope-Operator (*`::`*) zugegriffen werden.
- Elementfunktionen sollen prinzipiell in der Implementierungsdatei (*`.cpp`*) implementiert werden. Dem Funktionsnamen muss dabei der Klassenname gefolgt von *`::`* vorangestellt werden. (Beispiel: *`int Stack::pop()`*)

7.5.1 Klassifizierung von Elementfunktionen

- Konstruktoren / Destruktoren
  - Konstruktor: erzeugen eines Objekts
  - Destruktur: vernichten, freigeben eines Objekts
- Modifikatoren
  - ändern den Zustand eines Objekts (Attribute ändern)
- Selektoren
  - greifen nur lesend auf ein Objekt zu (immer *`const`* definieren!)
  - Beispiel: *`bool Stack::isEmpty() const;`*
- Iteratoren
  - Erlauben, auf Elemente eines Objekts in einer definierten Reihenfolge zuzugreifen

7.5.2 *inline*-Funktionen

- Elementfunktionen, die innerhalb der Deklaration der Klassenschnittstelle (im *`.h`*-File) implementiert sind, werden als (implizite) *`inline`* - Funktionen behandelt.
- Implizite *`inline`* - Funktionen verletzen zwar das Information Hiding Prinzip und sollten deshalb grundsätzlich vermieden werden.
- Jedoch: die impliziten *`inline`* - Funktionen sind die Funktionen, die garantiert immer *`inline`* verwendet werden (mit einigen wenigen Ausnahmen).
- Elementfunktionen können in der Klassenimplementation explizit mit dem Schlüsselwort *`inline`* gekennzeichnet werden.

7.5.3 mutable - Attribut

Ein Datenelement, das nie *const* werden soll (auch nicht bei *const*-Elementfunktionen) kann mit *mutable* gekennzeichnet werden.

```
class Stack
{
  public:
    int pop();
    int peek() const;
    bool isEmpty() const;
  private:
    int elem[maxElems];
    int top;
    mutable bool error;
};

int Stack::peek() const
{
  error = top == 0;
  if (!error)
    elem[top-1];
  else
    return 0;
}
```

7.5.4 const - Elementfunktion

- Elementfunktionen, die den Zustand eines Objekts nicht ändern (Selektoren) sollen explizit mit dem Schlüsselwort *const* gekennzeichnet werden.
- Das Schlüsselwort *const* muss sowohl im Prototypen als auch in der Implementierung geschrieben werden.

```
bool Stack::isEmpty() const;
...
bool Stack::isEmpty() const
{
  return top == 0;
}
```

7.6 static - Klassenelemente

- Grundsätzlich besitzt jedes Objekt einer Klasse seine eigene private Instanz aller Attribute einer Klasse.
- Wenn ein Attribut mit *static* gekennzeichnet wird, dann teilen sich alle Objekte dieser Klasse eine einzige Instanz dieses Attributs, d.h. ein statisches Attribut ist nur einmal für alle Objekte einer Klasse im Speicher vorhanden.
- *static* - Elemente befinden sich ausserhalb eines Objektkontexts.
- *static* - Elemente können auch über den Klassennamen angesprochen werden (da sie sich im Kontext einer Klasse befinden).

7.7 *this* - Pointer

Der *this*-Pointer ist ein Pointer auf das eigene aktuelle Objekt, welches eine Methode aufgerufen hat.

```
AnyClass& AnyClass::aMethod(const AnyClass& obj)
{
  this->anyFoo();
  if (this == &obj)    // testen, ob eigene Adresse gleich
    ...                // Adresse von obj ist
  return *this;        // eigenes Objekt zurückgeben
}
```

7.8 Konstruktor (am Beispiel der Klasse TString)

7.8.1 Aufgaben des Konstruktors

- die Neugründung eines Objekts einer Klasse
- das saubere Initialisieren des Objekts, d.h. alle Attribute des Objekts müssen auf einen definierten Wert gesetzt werden
- Der Konstruktor hat in *C++* denselben Namen wie die Klasse, hat keinen Rückgabetyp (auch nicht *void*) und kann überladen werden. Beispiel: *Stack::Stack();*

7.8.2 Aufruf des Konstuktors

- Der Konstruktor soll nie explizit aufgerufen werden.
- Der Konstruktor wird vom System automatisch (implizit) aufgerufen, wenn ein Objekt erzeugt wird: *Stack s;*
- Wenn durch den *new*-Operator Speicher angefordert **und** erhalten wird, dann wird der Konstruktor vom System ebenfalls automatisch aufgerufen: *Stack* pS = new Stack;*

7.8.3 Default-Konstruktor

- Der Default-Konstruktor ist der Konstruktor ohne Parameter.
- Er wird vom System automatisch erzeugt, wenn für eine Klasse kein Konstruktor explizit definiert ist.
- Der Default-Konstruktor kann auch selbst definiert werden.
  - Das ist insbesondere dann notwendig, wenn innerhalb des Objekts Speicher dynamisch alloziert werden muss (bei der Objekterzeugung).

```
class TString
{
  public:
    TString();
    int getLen() const;
  private:
    int len;
    char*  str;
};
```

7.8.4 Implementation/Initialisierung

Die Definition der Attribute muss der Reihe nach erfolgen, so wie im Headerfile. Dabei gibt es die folgenden zwei Möglichkeiten:

7.8.4.1 mittels Anweisung

```
TString::TString()
{
  len = 0;
  str = 0;
}
```

7.8.4.2 mittels Initialisierungsliste

```
TString::TString()
  : len(0), str(0)
{
}
```

Objektinitialisierungen werden, sofern dies möglich ist, über die Initialisierungsliste des Konstruktors und nicht im Anweisungsteil durchgeführt. (Effizienzgründe)

7.8.5 Überladen von Konstruktoren

Objekterzeugung:

```
TString str = "Hello World"; // implicit call
TString str = TString("Guten Morgen"); // explicit call
```

Deklaration:

```
public:
    TString();
    TString(const char* p);
```

7.8.6 Konstruktoren und Function Casts

Bei nur einem Parameter kann ein Konstruktor auch zur Typumwandlung benutzt werden (explicit call).

7.8.7 Explizite Konstruktoren

Falls das implizite Aufrufen eines Konstruktors nicht erwünscht ist, kann er mit *explicit* gekennzeichnet werden. Damit kann dieser Konstruktor nicht mehr implizit, sondern nur explizit aufgerufen werden.

```
explicit TString(int nr);
```

7.8.8 Copy-Konstruktor

- Der Copy-Konstruktor wird dazu verwendet, Objekte zu kopieren.
- Er erhält als Parameter immer eine konstante Referenz auf ein Objekt der Klasse.
- Wenn ein Objekt Speicher auf dem Heap alloziert, muss ein eigener Copy-Konstruktor definiert werden.

Der Copy-Konstruktor wird automatisch aufgerufen, wenn ...

- ... ein Objekt mit einem anderen Objekt derselben Klasse initialisiert wird.
- ... ein Objekt als Wertparameter (by value) an eine Funktion übergeben wird (nicht aber bei Referenzparametern).
- ... ein Objekt by value als Resultat einer Funktion zurückgegeben wird (nicht bei Referenzrückgabewerten).

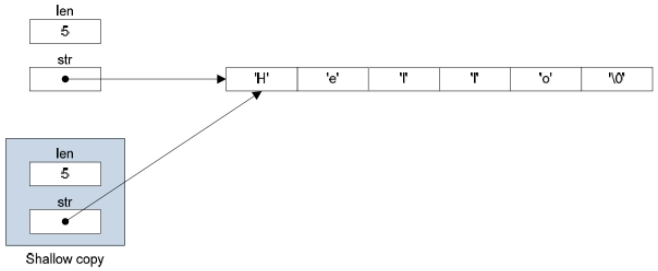
Folgend ein Beispiel, wie ein eigener C-tor im Headerfile implementiert wird:

```
TString(const TString& s);
```

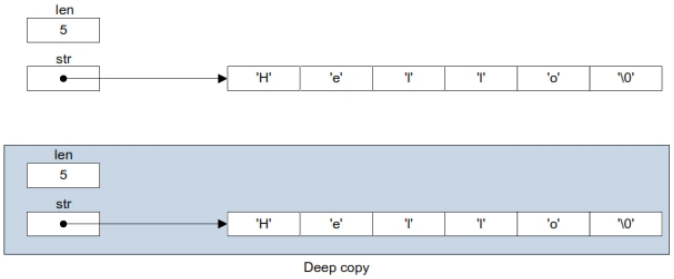
7.8.8.1 Shallow Copy vs. Deep Copy

- Wenn für eine Klasse kein Copy-Konstruktor definiert wird, erzeugt das System einen Standard-Copy-Konstruktor.
- Dieser kopiert alle Datenelemente (memberwise assignment). Bei Pointern, welche auf den Heap zeigen, wird nur die Adresse kopiert, nicht aber der Speicher auf dem Heap. Man nennt das shallow copy. (shallow = flach).
- Bei einer deep copy werden auch die Speicherbereiche, auf welche die Pointer zeigen, kopiert. Diese deep copy muss in einem selbst definierten Copy-Konstruktor implementiert werden.

7.8.8.2 Shallow-Copy



7.8.8.3 Deep-Copy



7.9 Destruktor

- Vollständige und ßaubereSZerstörung eines nicht mehr benötigten Objekts
- Sie werden automatisch aufgerufen, wenn der Gültigkeitsbereich des definierten Objekts ausläuft
- Die häuftigste Aufgabe ist die Freigabe von nicht mehr benötigten Speicher auf dem Heap
- sehr häufig (Wenn kein Speicher auf dem Heap vorhanden ist) wird kein Destruktor definiert, da das System dann automatisch aufräumt
- Destruktoren haben keine Argumente und keine Rückgabetypen
- Die Reihenfolge des Aufrufs der Destruktoren ist umgekehrt wie die der Konstruktoren. (das zuletzt erzeugte Objekt wird zuerst aufgeräumt)
- sobald eine Methode mit *virtual* gekennzeichnet ist, muss der Destruktor auch virtual sein.

Folgend ein Beispiel, wie ein eigener D-tor im Headerfile implementiert wird:

```
~TString()
```

7.10 Member In-Class Initialization

- Ab $C++11$ können bei der Klassendeklaration den Attributen Initialisierungswerte zugewiesen werden.
- Implementationstechnisch sind sie äquivalent zur Initialisierungsliste bei C-tors

```
class TString
{
  public:
    TString(); // Default-Constructor
  private:
    int len = 0;
    char* str = nullptr;
};
```

7.11 kanonische Form von Klassen

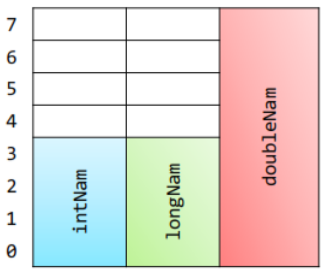
- Die kanonische Form ist die Form, die es erlaubt, eine Klasse wie einen «normalen» Datentyp zu benutzen (ist für alle Klassen anzustreben).
- 3 Bedingungen:
  1. Ein korrekter Default-Konstruktor, plus evtl. weitere Konstruktoren
  2. Bei dynamischen Daten, braucht es einen Zuweisungsoperator (später) und einen Copy-Konstruktor
  3. Ein (virtueller) Destruktor garantiert die korrekte Zerstörung von Objekten

7.12 Unions

- ähnlich wie Struktur
- im Gegensatz zur Struktur ist aber nur ein einziges Feld jeweils aktiv (abhängig vom Typ)
- Die Grösse einer Union ist so gross wie das grösste Feld der Union
- **Vorsicht**, der Programmierer muss verfolgen, welcher Typ jeweils in der Union gespeichert ist. Der Datentyp, der entnommen wird, muss der sein, der zuletzt gespeichert wurde.

7.12.1 Beispiel

```
union Vario
{
  private:
    int intNam;
    long longNam;
    double doubleNam;
}
```



7.13 Bitfelder

- Innerhalb eines int?s können einzelne Bitgruppen definiert und angesprochen werden
- Kann nützlich sein, wenn auf einzelne Bitgruppen in einem Register zugegriffen werden soll
- **Achtung:** ist nicht definiert, ob die Bits von links oder von rechts aufgefüllt werden (Compilerabhängig)
- Sollte also nicht eingesetzt werden, wenn der Code portabel sein soll
- Der Datentyp eines Bits ist immer derselbe, wie das ganze Bitfeld gross ist

```
struct FieldName
{
  unsigned int a: 3; // definiert 3 Bits fuer a
  unsigned int b: 4; // definiert die naechsten 4 Bits fuer b
}
```

7.14 Überladen von Operatoren

Operatoren (z.B. +, ==, etc.) können wie Funktionen überladen werden.

7.14.1 Überladbare Operatorfunktionen in C++

- `new`
- `delete`
- `new[ ]`
- `delete[ ]`
- `+`
- `-`
- `*`
- `/`
- `%`
- `^`
- `&`
- `|`
- `~`
- `!`
- `=`
- `<`
- `>`
- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `^=`
- `&=`
- `|=`
- `«`
- `»`
- `»=`
- `«=`
- `==`
- `!=`
- `<=`
- `>=`
- `&&`
- `||`
- `++`
- `-`
- `,`
- `->*`
- `->`
- `()`
- `[ ]`

7.14.2 Randbedingungen

- Die Anzahl der Operanden (Argumente) muss gleich sein wie beim ursprünglichen Operator.
- Die Priorität des überladenen Operators kann nicht ändern.
- Neue Operatoren können nicht eingeführt werden.
- Default-Argumente sind bei Operatoren nicht möglich.

7.14.3 Operator Overloading als Elementfunktion

Der neu definierte Operator wird als Elementfunktion implementiert. Damit ist der Zugriff auf `private` und `protected` Attribute der Klasse möglich.

```
class TString
{
  ...
  bool operator <(const TString& s) const;
  ...
};
bool TString::operator <(const TString& s)
  const
{
  ...
}
```

Achtung: Zwingend als Elementfunktion zu implementieren sind:
Zuweisungsoperator `=`, Indexaufruf `[ ]`, Funktionsaufruf `()` und Zeigeroperator `->`

7.14.4 Operator Overloading als normale Funktion

Die Operatorfunktionen werden meist als normale Funktion implementiert. Dadurch besteht jedoch kein Zugriff mehr auf die `private` und `protected` Elemente der Klasse. Die Operatorfunktion muss deshalb als `friend` deklariert werden.

```
class TString
{
  ...
  friend bool operator <(const TString& s1,
    const TString& s2);
  ...
};
bool operator <(const TString& s1,
  const TString& s2)
{
  ...
}
```

8 Templates

8.1 Motivation

Wesentliche Vorteile von Templates sind:

- **Single-Source-Prinzip:** Für x Varianten derselben Datenstruktur existiert genau eine Version des Sourcecodes, der geändert und gewartet werden muss.
- **Höhere Wiederverwendbarkeit:** Klassen-Templates sind bei geeigneter Wahl ihrer Parameter allgemein einsetzbar und einfach wiederverwendbar.
- **Statische Bindung:** Die Bindung zur Übersetzungszeit hat in Bezug auf Typsicherheit und Fehlererkennung zweifellos grosse Vorteile gegenüber generischen C-Lösungen mit *void**-Zeigern, aber zum Teil auch gegenüber typisch objektorientierten Varianten wie sie zum Beispiel in Smalltalk üblich sind.
- **Dead Code:** Traditionelle Bibliotheken belegen Speicher unabhängig davon, ob eine einzelne Funktion wirklich verwendet wird. Dies kann zu Dead Code führen, d.h. zu Code, der niemals ausgeführt wird.

8.2 Funktions-Templates

- Templates verwenden den Typ als Variable.
- Die Algorithmen können unabhängig vom Typ (generisch) implementiert werden.
- Templates sind keine Funktionsdefinitionen, sie beschreiben dem Compiler nur, wie er den Code definieren soll, d.h. der Compiler nimmt den konkret verwendeten Typ, setzt diesen in das Template ein und compiliert den so erhaltenen Code.
- Die Bindung zum konkreten Typ geschieht bereits zur Compiletime (early binding), sobald bekannt ist, mit welchem Typ das Template aufgerufen (benutzt) wird.

8.2.1 Syntax

- Vor den Funktionsnamen wird das Schlüsselwort *template*, gefolgt von einer in spitzen Klammern eingeschlossenen Parameterliste gestellt.
- Die Parameterliste enthält eine (nicht leere) Liste von Typ- und Klassenparametern, die mit dem Schlüsselwort *class* oder *typename* beginnen. Die einzelnen Parameter werden mit Komma getrennt.

```
template<typename ElemType>
ElemType minimum(ElemType
    elemField[],
  int fieldSize);
template<typename A, typename B>
int foo(A a, B b, int i);
```

8.2.2 inline bei Templates

inline muss zwischen *template* und dem Returntyp stehen. Achtung: Bei Verwendung von *inline* speziell zusammen mit Templates besteht die Gefahr von Code Bloat.

8.2.3 Überladen

- Funktions-Templates können mit anderen Funktionstemplates und auch mit normalen Funktionen überladen werden.
- Namensauflösung:
  - Compiler geht Liste der möglicherweise passenden Funktions-Templates durch und erzeugt die entsprechenden Template-Funktionen.
  - Ergebnis ist eine Reihe von (eventuell) passenden Template-Funktionen, ergänzt durch die vorhandenen normalen Funktionen.
  - Aus dieser ganzen Auswahl wird die am besten passende Funktion ausgewählt.

8.2.4 Ausprägung

- Sobald ein Typ in einem Funktions-Template verwendet wird, erkennt der Compiler, dass es sich um ein Template handelt und prägt es für diesen Typ aus (implizite Ausprägung).
- Für die Auflösung werden nur die Funktionsparameter betrachtet, der Rückgabetyp wird nicht ausgewertet.

```
int iF[] = {1, 54, 34, 23, 67, 4};
int i = minimum(iF, sizeof(iF)/sizeof(iF
    [0]));
```

8.2.5 Explizite Qualifizierung

- Funktions-Templates können explizit mit einem Typ qualifiziert werden.

```
int iF[] = {1, 54, 34, 23, 67, 4};
int i = minimum<int>(iF, sizeof(iF)/sizeof
    (iF[0]));
```

8.3 Klassen-Templates

8.3.1 Definition

- Klassen-Templates sind mit Typen oder Konstanten parametrisierbare Klassen.
- Im Gegensatz zu Funktions-Templates können in Klassen-Templates auch die Attribute der Klassen mit variablen Typen ausgestattet sein.
- Ein Klassen-Template kann auch von Ausdrücken abhängig sein. Diese Ausdrücke müssen aber zur Compiletime aufgelöst werden können.

8.3.2 Syntax

- Die Syntax ist analog zu den Funktions-Templates.
- Vor die Klassendeklaration wird das Schlüsselwort *template*, gefolgt von einer in spitzen Klammern eingeschlossenen Parameterliste gestellt.
- Die Parameterliste enthält eine (nicht leere) Liste von Typ- und Klassenparametern, die mit dem Schlüsselwort *class* oder *typename* beginnen oder auch von Ausdrücken. Die einzelnen Parameter werden mit Komma getrennt.

```
// Deklaration
template<typename ElemType, int size=100>
class Stack
{
  public:
    Stack();
    ~Stack();
    void push(const ElemType& elem);
    ElemType pop();
    bool wasError() const;
    bool isEmpty() const;
  private:
    ElemType elems[size];
    int top;
    bool isError;
};

// Definition
template<typename ElemType, int size>
void Stack<ElemType, size>::push(const ElemType&
    elem)
{
}

// Nutzung
Stack<int, 10>  s1;
  // s1 ist ein Stack mit 10 int's
Stack<int>      s2;
  // s2 ist ein Stack mit 100 int's (Default)
Stack<double>   s3;
  // s3 ist ein Stack mit 100 double's
```

8.4 Klassen-Templates und getrennte Übersetzung

8.4.1 Möglichkeit 1

```
// stack.h
#ifndef STACK_H_
#define STACK_H_
template<class ElemType, int size>
class Stack
{
  ...
};

// ugly, but cannot be avoided
#include "stack.cpp"
#endif
```

```
// stack.cpp

// all definitions of class Stack
// do NOT #include "stack.h"
```

```
// client.cpp
#include "stack.h"

int main()
{
  Stack<int, 50> s;
  ...
  return 0;
}
```

Mögliche Probleme:
`stack.cpp` darf nicht "normal" compiliert werden. Eine IDE wie Eclipse nimmt allenfalls alle `*.cpp`-Files in ihren Compile-Prozess. `stack.cpp` muss davon ausgeschlossen werden.

8.4.2 Möglichkeit 2

```
// stack.t
#ifndef STACK_T_
#define STACK_T_
template<class ElemType, int size>
class Stack
{
  ...
};

// all definitions of class Stack

#endif
```

Sowohl die Deklaration als auch die Definition des Templates sind im File `stack.t` vorhanden. Dadurch gibt es keine Probleme mit IDE's.

Nachteil:
Deklaration und Definition sind nicht getrennt.

```
// client.cpp
#include "stack.t"

int main()
{
  Stack<int, 50> s;
  ...
  return 0;
}
```

Fazit:
Sowohl Variante #1 als auch #2 haben Vor- und Nachteile.

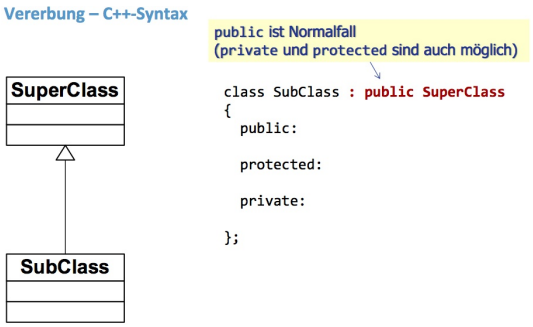
Wählen Sie eine aus!
(Ich bevorzuge #1)

9 Vererbung (Inheritance)

Vererbung ist ein Konzept, das es erlaubt, neue Klassen auf Basis von alten Klassen zu definieren. Die neuen (Unter-, Sub-) Klassen besitzen, ohne Eingriffe in den Sourcecode der bereits bestehenden (Ober-, Basis-, Super-) Klassen, all deren Eigenschaften, sie erben deren Verhalten und Daten. Den Vorgang der Vererbung nennt man auch Ableiten.

9.1 Einsatz der Vererbung

- Bestehende Klassen erweitern (zusätzliche Attribute und Elementfunktionen)
- Bestehende Methoden einer Basisklasse ändern (überschreiben)
- Einsatz nur wenn eine **IST-EIN** Beziehung besteht (z.B. Baum **ist eine** Pflanze, Blume **ist eine** Pflanze)



9.2 Ableiten einer Klasse

Der Syntax der Ableitung einer Klasse ist oben aufgeführt. Als weiteres Beispiel ist im Anhang das Beispiel des *ComicCharacters* und *SuperHero* eingefügt. *SuperHero* **ist ein** *ComicCharacter*.

- *friend*-Beziehungen werden nicht vererbt
- Ein Objekt einer Oberklasse kann Objekte einer beliebigen Unterklasse aufnehmen aber nicht umgekehrt (Substitutionsprinzip)
- Ein Objekt einer vererbten Klasse enthält alle Teile der Basisklasse und zusätzlich noch die spezifischen eigenen Teile.
- Das Objekt ist somit mindestens so gross wie jenes der Basisklasse (es gibt keine Vererbung *by reference*)

```
class SuperClass{};
class SubClass:public SuperClass
    {};
SuperClass super;
SubClass sub;
super = sub; // ok
sub = super; // geht nicht
```

9.3 Zugriff auf Elemente der Basisklasse

Bei Vererbung mit *public* (Normalfall):

- Zugriff möglich auf alle *public*- und *protected*- Elemente der Basisklasse, die Zugriffsrechte (*public*, *protected*) der Basisklasse werden in der abgeleiteten Klasse beibehalten

Bei Vererbung mit *protected*:

- Zugriff möglich auf alle *public*- und *protected*- Elemente der Basisklasse, die Zugriffsrechte von *public* und *protected* der Basisklasse werden in der abgeleiteten Klasse zu *protected*

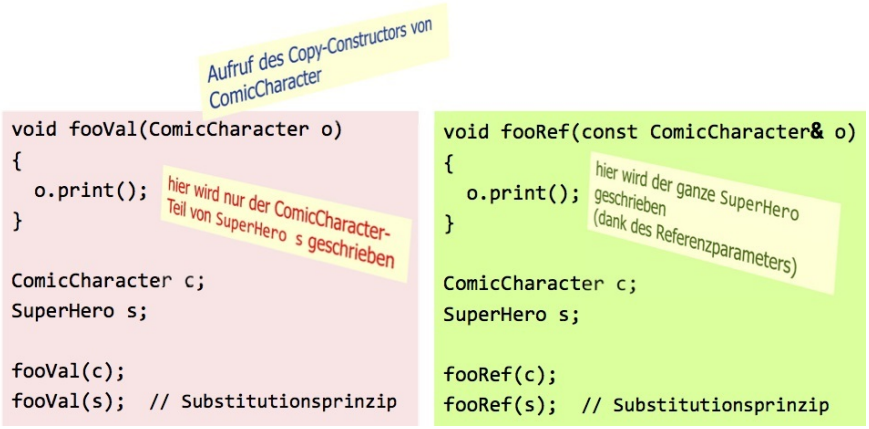
Bei Vererbung mit *private*:

- Zugriff möglich auf alle *public*- und *protected*- Elemente der Basisklasse, die Zugriffsrechte von *public* und *protected* der Basisklasse werden in der abgeleiteten Klasse zu *private*

Bei allen drei: kein Zugriff auf *private*-Elemente der Basisklasse

9.4 Slicing Problem

Links: Beim Kopieren werden nur die *ComicCharacter*-Teile berücksichtigt. Durch das Kopieren wird alles überflüssige weggeschnitten, übrig bleibt ein reines *ComicCharacter* Objekt im Fall von *s* führt dies dazu, dass die erweiterten *SuperHero* Daten und Funktionen verloren gehen.
Rechts: Hier wird dank des Referenzparameters der gesamte Superheld ausgegeben.



9.5 Vererbung und Gültigkeitsbereiche

Die Klasse *C* enthält alle Elemente von *B* und somit auch von *A*. *A* jedoch hat kein *i* und kann auch von keiner Oberklasse erben, dies ergibt den Fehler. *B* hat zwar auch kein *j*, erbt aber das von *A*.

```
class A
{
  public:
    int j;
};
```

```
class B : public A
{
  public:
    int i;
};
```

```
class C : public B
{
  public:
    void foo();
    int i;
    int j;
};
```

```
void C::foo()
{
 cout << i;      // C::i
 cout << j;      // C::j
 cout << B::i;   // B::i
 cout << B::j;   // A::j
 cout << A::i;   // Fehler
 cout << A::j;   // A::j
}
```

9.6 Elementfunktionen bei abgeleiteten Klassen

9.6.1 Konstruktoren

In einem Konstruktor müssen alle Elemente eines Objekts (auch die ererbten) initialisiert werden. Folgendes Beispiel zeigt die direkte Initialisierung aller Elemente. Vor allem bei grossen oder mehreren Klassen ist dies nicht zielführend. Stattdessen wird das Chaining Prinzip angewandt. Falls kein Aufruf eines Basislklassen-Konstruktors in der Initialisierungsliste eines Konstruktors erscheint, so fügt der Compiler automatisch den Default-Konstruktor der Basisklasse ein.

```
Book::Book(const string& aName,
           int aCode,
           double aPrice,
           int aRating,
           const string& aComment,
           const string& aAuthor,
           const string& aTitle,
           const string& aIsbn) :
author(aAuthor), title(aTitle), isbn(aIsbn) // eigene
   Attribute
{
  setName(aName);             // Attribute der
   Basisklasse
  setCode(aCode);
  setPrice(aPrice);
  setRating(aRating);
  setComment(aComment);
}
```

9.6.2 Initialisierung durch Chaining

Jede Klasse erledigt nur die eigenen Aufgaben. Aufgaben, die ererbte Methoden übernehmen können, werden diesen delegiert (Aufruf der jeweiligen Konstruktoren)
Wichtig: die Elemente der Basisklasse müssen immer als erste initialisiert werden

```
Book::Book(const string& aName,
           int aCode,
           double aPrice,
           int aRating,
           const string& aComment,
           const string& aAuthor,
           const string& aTitle,
           const string& aIsbn) :
Article(aName, aCode, aPrice, aRating, aComment),
author(aAuthor), title(aTitle), isbn(aIsbn)
{
}
```

9.6.3 Copy-Konstruktor

- Wenn kein Copy Constructor explizit definiert wird, so erzeugt das System einen
- Darin wird immer (ebenfalls automatisch) zuerst der Copy Constructor der Basisklasse aufgerufen

9.6.4 Destruktor

- Auch Destruktoren werden nach dem Chaining-Prinzip aufgebaut
- Jede Klasse kümmert sich um die eigenen Attribute und überlässt jene der Basisklasse auch der Basisklasse
- Destruktoren müssen nie explizit aufgerufen werden. Der Destruktor der Basisklasse wird **am Schluss** des Destruktors immer automatisch aufgerufen
  Ein leerer Destruktor der Art

```
~SuperHero();
```

  ruft automatisch den Basisklassen-Destrukor (von *ComicCharacter*) auf.

9.6.5 Überschreiben von ererbten Methoden

- Falls ererbte Methoden nicht das erfüllen, was eine bestimmte Klasse möchte, dann können diese Methoden neu definiert (überschrieben) werden.
- Methoden, welche in einer abgeleiteten Klasse überschrieben werden können, müssen in der Basisklasse mit *virtual* gekennzeichnet sein.
- Ab *C++11* sollen in der Unterklasse die überschriebenen Methoden mit *override* gekennzeichnet werden. Damit weiss der Compiler, dass mit dieser Methode eine andere überschrieben wird.
- Im Anhang wird dieses überschreiben einer Methode beim *SuperHero* für die Funktion *dance()* vorgenommen. Während ein normaler *ComicCharacter* tanzt, wird diese Funktion beim *SuperHero* überschrieben und mit tanzt nicht überschrieben.

10 Polymorphismus / Mehrfachvererbung / RTTI

Dieses Kapitel beschreibt die dynamischen objektorientierten Sprachmerkmale von *C++*. Erst durch diese wird *C++* zu einer echten objektorientierten Programmiersprache.

10.1 Polymorphismus

10.1.1 Problem

- Abstrakter Auftrag, aber die Ausführung ist bereits sehr konkret.
- Eine immer gleich heissende Elementfunktion hat unterschiedliche Implementationen, je nach Art des aktuellen Objekts.

10.1.2 static Binding

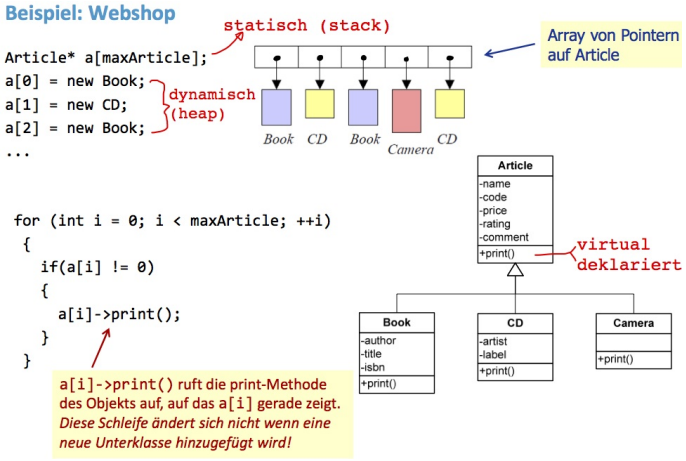
Ist der Normalfall und wird auch early Binding genannt. Hier wird bereits zur Compilezeit festgelegt, welcher Code ausgeführt wird.

10.1.3 dynamic Binding

- Erst zu Laufzeit wird in Abhängigkeit des Objekts festgelgt, welcher Code ausgeführt wird.
- Das ist das Konzept des Polymorphismus.
- Der mächtigste OO-Mechanismus (oft präziser mit run-time polymorphism bezeichnet)
- Regeln:
  - Eine Funktion soll dann virtual deklariert werden, wenn sie in der abgeileiteten Klasse neu definiert (überschieben) wird, sonst nicht!
  - Wenn mindestens eine Elementfunktion virtual ist, muss auch der Destruktor virtual sein.
  - Der Destruktor muss auch dann virtual sein, wenn ein Objekt einer Unterklasse dynamisch erzeugt wird und einem Pointer auf die Basisklasse zugewiesen wird (Substitutionsprinzip)

10.1.4 Beispiel

- Der statische Datentyp bezeichnet den Datentyp bei der Deklaration. Im Beispiel: `a` ist ein Array von Pointer auf `Article`
- Der dynamische Datentyp bezeichnet den effektiven Datentyp zur Laufzeit Im Beispiel: `a[0]` ist ein Pointer auf `Book`, `a[1]` ein Pointer auf `CD`, etc.



10.1.5 Polymorphe (virtuelle) Klassen

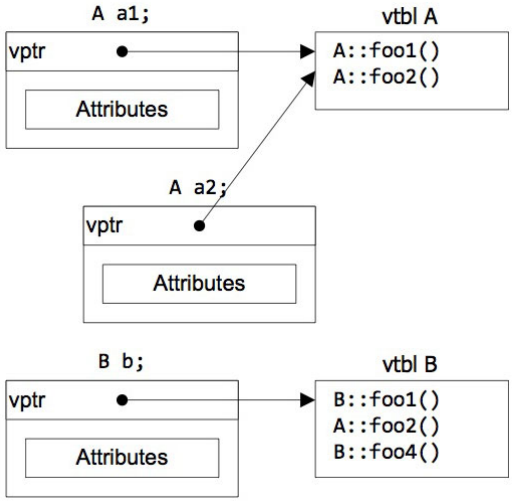
- Eine Klasse, welche mindestens eine virtuelle Funktion deklariert, heisst virtuell (polymorph)
- Virtuelle Klassen bewirken einen Mehraufwand für den Compiler und sind darum langsamer in der Ausführung
- Konstruktoren sind nie virtuell
- Destruktoren virtueller Klassen müssen immer als virtuell deklariert werden, sonst wird nur der Destruktor der Basisklasse aufgerufen
- **Nicht virtuelle Methoden dürfen nicht überschrieben werden** (könnten technisch gesehen, führt aber zu unüberschaubaren Fehlern)
- In der Unterklasse sollen die überschriebenen Methoden mit override gekennzeichnet werden. Dient der Übersicht und wenn die zu überschreibenden Methode in der Basisklasse nicht virtuell ist, ruft der Compiler aus.

10.1.6 Repräsentation polymorpher Objekte im Speicher

- In der Virtual Function Table (vtbl) vermerkt das System der Reihe nach die Adressen der für eine Klasse gültigen virtuellen Elementfunktionen
- Das System legt für jede polymorphe Klasse eine vtbl an
- Jedes Objekt einer polymorphen Klasse enthält einen Virtual Pointer vptr, welcher auf die vtbl der entsprechenden Klasse zeigt

```
class A
{
  public:
    virtual void foo1();
    virtual void foo2();
    void foo3();
};

class B : public A
{
  public:
    virtual void foo1();
    virtual void foo4();
};
```



10.2 Abstrakte Klassen

Eine abstrakte Klasse ist ein Klasse, die mehr oder weniger vollständig ist und dazu dient, Gemeinsamkeiten der abgeleiteten Klassen festzuhalten (z.B. `ComicCharacter`). `ComicCharacter` legt fest, dass alle Comicfiguren die Methoden `print()`, `dance()` und `sing()` verstehen.

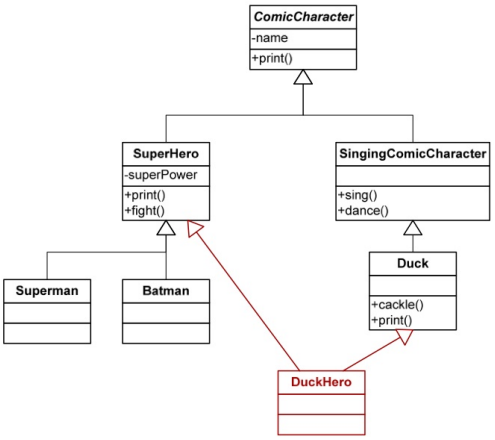
- Ein Kreis ist z.B. ein Spezialfall einer Ellipse. Es ist aber nicht sinnvoll, ihn so zu programmieren, da er sonst Eigenschaften erbt, die nicht verwendet werden
- Es wäre möglich, Kreis und Ellipse als zwei unabhängige Klassen zu programmieren. Dann müssten aber alle Eigenschaften, die diese gemeinsam haben, doppelt programmiert werden
- Dies versucht die objektorientierte Programmierung zu vermeiden
- Es ist besser, die Eigenschaften, die Kreise und Ellipsen gemein haben, in einer Basisklasse zu programmieren
- Die Kreis- und Ellipsenklasse erben dann parallel von der gemeinsamen Basisklasse
- Die Basisklasse ist aber unvollständig, es handelt sich um eine abstrakte Klasse
- Es können **keine** Objekte von abstrakten Klassen gebildet werden
- In `C++` können rein virtuelle Funktionen (pure virtual functions) deklariert werden, die in der Basisklasse nicht von einer Definition begleitet werden. Beispiel:

```
virtual double getArea() = 0;
virtual double getArea() = 0
{
...
}
```

- **Klassen, die mindestens eine rein virtuelle Funktion deklarieren, sind abstrakte Klassen**
- Ist eine Klasse erst einmal als abstrakt definiert, kann diese nur durch Vererbung vervollständigt und dadurch nutzbar gemacht werden

10.3 Mehrfachvererbung

Bei der Mehrfachvererbung wird eine Klasse von mehreren Basisklassen abgeleitet. So kann z.B. eine Klasse `DuckHero` definiert werden, die sowohl von `SuperHero` als auch von `SingingComicCharacter` erbt.



Guter Einsatz der Mehrfachvererbung ist, wenn alle ausser höchstens einer Basisklasse ausschliesslich aus rein virtuellen Funktionen bestehen (Interfaces). Die neue Klasse implementiert dann die aufgelisteten Interfaces.
Das obige Beispiel ist im Anhang unter 15.4 angehängt.

Der Syntax bei der Mehrfachvererbung lautet wie folgt (Basisklassen durch Komma getrennt). Dabei müssen die Konstruktoren der Basisklassen in der Ordnung gelistet werden, in der sie aufgerufen werden sollen:

```
class DuckHero: public Duck, public SuperHero
{
  public:
    DuckHero(const std::string& aName = "",
             const std::string& aPower = "noPower");
    virtual ~DuckHero();
    virtual void print() const;
};
```

Durch die Mehrfachverbung treten oft Probleme auf. Problem 1 ist jenes der Mehrdeutigkeit von Methoden. Im Fall von `print()` ergeben sich mehrere Möglichkeiten. Um die Mehrdeutigkeit zu umgehen, muss der Gültigkeitsbereich angegeben werden:

```
DuckHero dh;
dh.print(); // Fehler: mehrdeutig!!
dh.Duck::print() // ok
```

Oder noch besser:

```
virtual void DuckHero::print
{ //bessere Loesung
  Duck::print();
}
```

Das Problem 2 ist das von mehrfachen Basisklassen (linker und rechter Baum). `DuckHero` ist von `Duck` und `SuperHero` abgeleitet und beinhaltet somit zwei `ComicCharacter`-Teile. Diese Mehrdeutigkeit kann durch virtuelles erben verhindert werden. Dies muss jedoch bereits eine Stufe höher geschehen.

10.4 RTTI (Laufzeit-Typinformation)

RTTI (Run-Time Type Information) ist die Möglichkeit den Typ eines Objekts einer polymorphen Klasse festzustellen. Er steht ausschliesslich für polymorphe Klassen zur Verfügung und sollte sehr zurückhaltend eingesetzt werden. Der RTTI-Mechanismus besteht im Wesentlichen aus zwei Operatoren und einer Struktur:

- Operator *dynamic_cast*
- Operator *typeid*
- Klasse *type_info*

10.4.1 Operator dynamic_cast

Syntax:

```
dynamic_cast<SuperHero*>(p)
```

- Versucht, den Zeiger *p* in einen Zeiger auf ein Objekt des Typs *SuperHero* umzuwandeln
- Der dynamische Datentyp von *p* ist massgebend
- Umwandlung wird dann durchgeführt, wenn *p* tatsächlich auf ein Objekt vom Typ *SuperHero*, bzw. auf eine davon abgeleitete Klasse zeigt.
- Andernfalls ist das Resultat der Umwandlung der Nullpointer!

10.4.2 Operator typeid

- Ermitteln des dynamischen Datentyps eines polymorphen Objekts
- Ergibt eine Referenz auf ein Objekt des Typs *type_info*. Diese Klasse beinhaltet u.a. eine Methode *name()*, welche den Namen der Klasse zurückgibt. Beispiel:

```
cout << "p ist ein " << typeid(*p).name() << "Objekt";
```

10.4.3 Struktur type_info

Die Struktur muss eingebunden werden

```
#include <typeinfo>
```

Sie bietet mind. folgende Funktionalität:

- die Operatoren *==* und *!=*
- die Methode *before*
- die Methode *name* (siehe Beispiel oben)

11 Assertions

Um Assertions überhaupft nutzen zu können muss zuerst die entsprechende Bibliothek eingebunden werden:

```
#include <cassert>
```

Assertions dienen zur Überprüfung von logischen Annahmen während der Entwicklungsphase, speziell für die Überprüfung von Anfangs- und Endbedingungen in einer Funktion. Wenn die Bedingung *false* ist, bricht das Programm ab. Es darf aber kein Nebeneffekt hineinprogrammiert werden, da asserts in der Release-Funktion wirkungslos sind.
Beispiel:

```
assert(i>0);
```

11.1 *static_assert*

Diese Art von Assertions werden bereits zur Compilezeit überprüft und man kann eine Textmeldung ausgeben falls Bedingung *false* ist. Da die Bedingung zwingend zur Compilezeit auswertbar sein muss, darf sie keine Variablen o.ä. beinhalten. Werden vorallem im Zusammenhang mit Templates genutzt.
Beispiel:

```
static_assert(sizeof(int) <= 4, "max 32 Bit werden unterstuetzt");
```

12 Exception Handling

Abnormale aber vorhersebare und mögliche Bedingung bei der Programmausführung.

12.1 Handling Strategie von System Exceptions

- In *Java* und *C#* gelangen die System Exceptions in die Sprache, d.h. eine LowLevel Exception wird in eine Exception der Programmiersprache gemappt.
- Die Sprache *C++* betreibt kein solches Exception Mapping, d.h. Low-Level Exceptions werden nicht von *C++* geworfen und können auch nicht mit *catch(...)* abgefangen werden.
- Der Hauptgrund dafür ist einmal mehr Effizienz. Wenn ständig Exceptions herumfliegen (auch wenn sie nicht abgefangen werden), dann beeinträchtigt das die Performance.
- Einzelne Systemumgebungen betreiben dennoch Exception Mapping in *C++* (z.B. *Microsoft* in *Visual C++*).

12.2 Ziel

- Der Normalfall soll einfach gelesen werden können
- Der Ausnahmefall ist klar und einfach geregelt
- Der Overhead soll möglichst klein sein
- Die Weiterreichung an die nächsthöhere Funktion im Call Stack soll einfach sein

12.3 Exceptionhandling in *C++*

- Exceptions werden in Form eines Objekts am Ort ihres Auftretens ausgeworfen (explizit oder auch äutomatisch").
- Exception Handler versuchen, diese Exception-Objekte aufzufangen.

12.3.1 Auslösen (Werfen) von Ausnahmen

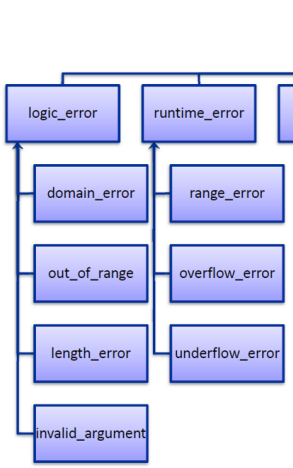
- Ausnahmen können mit dem Schlüsselwort *throw* explizit ausgeworfen werden.
- Nach einem *throw*-Befehl wird das Programm abgebrochen und beim ersten passenden umgebenden Handler fortgesetzt.
- Dabei werden alle lokalen Objekte wieder automatisch zerstört (Stack unwinding).
- Geworfen werden kann ein beliebiges Objekt (üblich: ein spezifisches *C++*-Ausnahmeobjekt).
- (Ausschliesslich) innerhalb eines Exception Handlers ist auch die Form *throw;* erlaubt. Dadurch wird die Exception an den nächsten Handler weitergereicht (Exception propagation).

12.3.2 Syntax

```
try
{
... // Code, der eine Exception auswerfen
    koennte
}
catch (const MyExceptionClass& exc)
{
... // wenn ein Objekt der Klasse
    MyExceptionClass oder einer Unterklasse
// davon ausgeworfen wurde, dann kann
    dieser Handler das Objekt fangen
}
// u.U. weitere Catches
```

12.3.3 Exception-Hierarchie in *C++* und ihre Headers

Ausnahmeobjekte können beliebigen Typs sein (z.B. auch *int*). Meist werden jedoch spezifische hierarchisch organisierte Ausnahmeklassen verwendet.



Exception	Header
exception bad_exception	<exception>
logic_error domain_error out_of_range length_error invalid_argument	<stdexcept>
runtime_error range_error overflow_error underflow_error	<stdexcept>
bad_cast bad_typeid	<typeinfo>
bad_alloc	<new>
ios_base::failure	<ios>

12.3.4 Laufzeit- vs. Logische Fehler

- Logische "Fehler"(logic_error)
  - Ausnahmen im Programmablauf, die bereits zur Entwicklungszeit ihre Ursache haben.
  - Theoretisch könnten diese Ausnahmen verhindert werden.
- Laufzeit "Fehler"(runtime_error)
  - Nicht vorhersehbare Ausnahmen wie z.B. arithmetische Überläufe.
  - Diese Ausnahmen treten erst zur Laufzeit auf, z.B. durch eine nicht erlaubte Benutzereingabe.

12.3.5 Mehrere Catches

- Ein oder mehrere Exception Handler können hintereinander definiert werden.
- Die einzelnen *catch*-Handler müssen sich in den Parametern unterscheiden.
- Wenn eine Exception geflogen kommt, wird der erste passende Handler genommen. Ein passender Handler macht ein *catch* auf genau diese Exception oder auf eine Basisklasse derselben.
- Deshalb (sehr wichtig): Der allgemeinste Handler (am meisten oben in der Hierarchie) muss als letzter definiert werden.
- Wenn kein Handler passt, dann wird im Aufrufstack nach oben gesucht, ob ein passender Handler vorhanden ist.
- Wenn auch dort keiner gefunden wird, dann wird die Funktion *terminate()* aufgerufen.
- *terminate()* beendet das Programm, kann aber auch selbst definiert werden.
- Catch all: Der folgende Handler fängt ausnahmslos alle Exceptions ab (und muss wenn gewünscht deshalb immer als letzter aufgeführt werden):

```
catch(...)
{
   ...
}
```

12.3.5.1 Exception Specification

void foo() throw(/ Liste der Exceptions */);*

- Liste beschreibt, welche Exceptions von einem Aufrufer von *foo()* erwartet werden müssen.
- Aber: garantiert auch, dass das Programm abstürzt, wenn eine andere als die spezifizierten Exceptions ausgeworfen wird, d.h. *foo()* muss dafür sorgen, dass wirklich nur die aufgelisteten Exceptions ausgeworfen werden.
- Genauer: falls eine nicht spezifizierte Exception ausgeworfen wird, dann wird die Funktion *unexpected()* aufgerufen, welche üblicherweise das Programm abbricht.
- *unexpected()* kann selbst definiert werden.
- **Ab *C++11* gilt jedoch:**
  - Exception Specifications sind deprecated (sollen nicht mehr verwendet werden)
  - Aber: dafür wurde ein neues Schlüsselwort *noexcept* eingeführt, um anzugeben, dass eine Funktion keine Exceptions auswirft (na ja)
  - *noexcept* ist auch ein Operator, dem als Argument ein Funktionspointer übergeben werden kann
    - returns *true*, falls Funktion mit *noexcept* spezifiziert ist
    - sonst *false*

Beispiele:

```
void foo1() throw(specificXcpt1, specificXcpt2);
// die zwei angegebenen Exceptions muessen vom Aufrufer von
// foo1() erwartet werden.

void foo2() throw();  // KEINE Exceptions koennen geflogen kommen

void foo3();  // beliebige Exceptions muessen erwartet werden
```

Beispiele ab *C++11*:

```
void foo1() throw(specificXcpt1, specificXcpt2); // deprecated
void foo2() throw(); // deprecated

void foo2() noexcept; // ab C++11 // KEINE Exceptions koennen geflogen kommen

void foo3();  // beliebige Exceptions muessen erwartet werden

bool a = noexcept(foo2); // returns true
bool b = noexcept(foo3); // returns false
```

13 Beispiele

13.1 Stack als Klasse

```
// Datei: Stack.h
// Schnittstellendefinition für Stack
// R. Bonderer, 26.03.2019

#ifndef STACK_H_
#define STACK_H_

class Stack
{
  public:
    Stack();
    // Default Ctor, initialisiert den Stack

    void push(int e);
    // legt ein Element auf den Stack, falls der Stack noch nicht voll ist
    // wasError() gibt Auskunft, ob push() erfolgreich war

    int pop();
    // nimmt ein Element vom Stack, falls der Stack nicht leer ist
    // wasError() gibt Auskunft, ob pop() erfolgreich war

    int peek() const;
    // liest das oberste Element vom Stack, falls der Stack nicht leer ist
    // wasError() gibt Auskunft, ob peek() erfolgreich war

    bool isEmpty() const;
    // return: true: Stack ist leer
    //         false: sonst

    bool isFull() const;
    // return: true: Stack ist voll
    //         false: sonst

    bool wasError() const;
    // return: true: Operation war fehlerhaft
    //         false: sonst

  private:
    Stack(const Stack& s);  // verhindert das Kopieren von Stack-Objekten
    enum {maxElems = 10};   // Anzahl Stackelemente
    int elem[maxElems];     // Array fuer Speicherung des Stacks
    int top;                // Arrayindex des naechsten freien Elements
    mutable bool error;     // true: Fehler passiert; false: sonst
    // mutable: auch const-Methoden können dieses Attribut setzen
};

#endif // STACK_H_
```

```cpp
// Datei: Stack.cpp
// implementiert Stackoperationen
// R. Bonderer, 26.03.2019

#include "Stack.h"

Stack::Stack()
  : top(0), error(false)
{
}

void Stack::push(int e)
{
  error = isFull();
  if (!error)
  {
    elem[top] = e;
    ++top;
  }
}

int Stack::pop()
{
  error = isEmpty();
  if (!error)
  {
    --top;
  }
  return elem[top];
}

int Stack::peek() const
{
  error = isEmpty();
  if (!error)
    return elem[top-1];
  else
    return elem[top];
}

bool Stack::isEmpty() const
{
  return top == 0;
}

bool Stack::isFull() const
{
  return top == maxElems;
}

bool Stack::wasError() const
{
  return error;
}
```

13.2 Stack als Template

```
/*
 * stack.h
 *
 *  Created on: 21.05.2013
 *      Author: rbondere
 */

#ifndef STACK_H_
#define STACK_H_

template<typename ElemType, int size = 10>
class Stack
{
  public:
    Stack();
    // Default-Konstruktor

    void push(const ElemType& e);
    // legt ein Element auf den Stack, falls der Stack noch nicht voll ist
    // wasError() gibt Auskunft, ob push() erfolgreich war

    ElemType pop();
    // nimmt ein Element vom Stack, falls der Stack nicht leer ist
    // wasError() gibt Auskunft, ob pop() erfolgreich war

    ElemType peek() const;
    // liest das oberste Element vom Stack, falls der Stack nicht leer ist
    // wasError() gibt Auskunft, ob peek() erfolgreich war

    bool isEmpty() const;
    // return: true: Stack ist leer
    //         false: sonst

    bool wasError() const;
    // return: true: Operation war fehlerhaft
    //         false: sonst

  private:
    ElemType elems[size]; // Speicher für Speicherung des Stacks
    int top;              // Arrayindex des naechsten freien Elements
    mutable bool error;   // true: Fehler passiert; false: sonst
    // mutable: auch const-Methoden können dieses Attribut setzen
};

// ugly include
#include "stack.cpp"
#endif // STACK_H_
```

```
/*
 * stack.cpp
 *
 *  Created on: 21.05.2013
 *      Author: rbondere
 */

template<typename ElemType, int size>
Stack<ElemType, size>::Stack() :
  top(0), error(false)
{
}

template<typename ElemType, int size>
void Stack<ElemType, size>::push(const ElemType& e)
{
  error = top == size;
  if (!error)
  {
    elems[top] = e;
    top++;
  }
}

template<typename ElemType, int size>
ElemType Stack<ElemType, size>::pop()
{
  error = top == 0;
  if (!error)
  {
    --top;
    return elems[top];
  }
  else
    return 0;
}

template<typename ElemType, int size>
ElemType Stack<ElemType, size>::peek() const
{
  error = top == 0;
  if (!error)
    return elems[top - 1];
  else
    return 0;
}

template<typename ElemType, int size>
bool Stack<ElemType, size>::isEmpty() const
{
  return top == 0;
}

template<typename ElemType, int size>
bool Stack<ElemType, size>::wasError() const
{
  return error;
}
```

13.3 Vererbung Comiccharacter

```
/*
 * ComicCharacter.h
 *
 *  Created on: 20.04.2009
 *      Author: rbondere
 */

#ifndef COMICCHARACTER_H_
#define COMICCHARACTER_H_

#include <string>

class ComicCharacter
{
  public:
    ComicCharacter(const std::string& aName = "");
    virtual ~ComicCharacter();
    virtual void print() const;
    virtual void dance() const;
    virtual void sing() const;
    void setName(const std::string& aName);
    const std::string& getName() const;
  private:
    std::string name;
};

#endif /* COMICCHARACTER_H_ */
```

```
/*
 * ComicCharacter.cpp
 *
 *  Created on: 20.04.2009
 *      Author: rbondere
 */

#include <iostream>
#include <string>
#include "ComicCharacter.h"

using namespace std;

ComicCharacter::ComicCharacter(const string& aName) :
  name(aName)
{
}

ComicCharacter::~ComicCharacter()
{
}

void ComicCharacter::print() const
{
  cout << "Name of ComicCharacter: " << name << endl;
}

void ComicCharacter::dance() const
{
  cout << name << " dances" << endl;
```

```
}

void ComicCharacter::sing() const
{
  cout << name << " sings" << endl;
}

void ComicCharacter::setName(const string& aName)
{
  name = aName;
}

const string& ComicCharacter::getName() const
{
  return name;
}
```

```
/*
 * SuperHero.h
 *
 *  Created on: 24.04.2013
 *      Author: rbondere
 */

#ifndef SUPERHERO_H_
#define SUPERHERO_H_

#include <string>
#include "ComicCharacter.h"

class SuperHero: public ComicCharacter
{
  public:
    SuperHero(const std::string& aName = "", const std::string& thePower = "noPower");
    virtual ~SuperHero();
    virtual void dance() const override;
    virtual void fight() const;
    const std::string& getSuperPower() const;
    void setSuperPower(const std::string& thePower);
  private:
    std::string superPower;
};

#endif /* SUPERHERO_H_ */
```

```
/*
 * SuperHero.cpp
 *
 *  Created on: 24.04.2013
 *      Author: rbondere
 */

#include <iostream>
#include <string>
#include "SuperHero.h"
using namespace std;

SuperHero::SuperHero(const string& aName,
                     const string& thePower) :
    ComicCharacter(aName), superPower(thePower)
```

```
{
}

SuperHero::~SuperHero()
{
}

void SuperHero::fight() const
{
  cout << getName() << " fights" << endl;
}

void SuperHero::dance() const
{
  cout << "Superheroes don't dance!" << endl;
}

void SuperHero::setSuperPower(const string& thePower)
{
  superPower = thePower;
}

const string& SuperHero::getSuperPower() const
{
  return superPower;
}
```

```
/*
 * ComicTest.cpp
 *
 *  Created on: 24.04.2013
 *      Author: rbondere
 */
#include <string>
#include <iostream>
#include "ComicCharacter.h"
#include "SuperHero.h"
using namespace std;

int main()
{
  ComicCharacter cc("Roadrunner");
  cc.sing();
  cc.dance();

  ComicCharacter* pcc = new ComicCharacter;
  pcc->setName("Tom");
  pcc->sing();
  pcc->dance();

  SuperHero sh("Lucky Luke", "Speed");
  sh.fight();
  cout << "Power of " << sh.getName() << " is " << sh.getSuperPower() << endl;
  sh.dance();

  delete pcc;
  return 0;
}
```

13.4 Mehrfachvererbung Comiccharacter

```
/*
 * ComicCharacter.h
 *
 *  Created on: 30.04.2013
 *      Author: rbondere
 */

#ifndef COMICCHARACTER_H_
#define COMICCHARACTER_H_

#include <string>

class ComicCharacter
{
  public:
    ComicCharacter(const std::string& aName = "");
    virtual ~ComicCharacter();
    virtual void print() const = 0;
    void setName(const std::string& aName);
    const std::string& getName() const;
  private:
    std::string name;
};

#endif /* COMICCHARACTER_H_ */
```

```
/*
 * ComicCharacter.cpp
 *
 *  Created on: 30.04.2013
 *      Author: rbondere
 */

#include <iostream>
#include <string>
#include "ComicCharacter.h"
using namespace std;

ComicCharacter::ComicCharacter(const string& aName) :
  name(aName)
{
}

ComicCharacter::~ComicCharacter()
{
}

void ComicCharacter::setName(const string& aName)
{
  name = aName;
}

const string& ComicCharacter::getName() const
{
  return name;
}
```

```
/*
 * SuperHero.h
 *
 *  Created on: 25.04.2019
 *      Author: rbondere
 */

#ifndef SUPERHERO_H_
#define SUPERHERO_H_

#include <string>
#include "ComicCharacter.h"

class SuperHero: virtual public ComicCharacter
{
  public:
    SuperHero(const std::string& aName = "",
              const std::string& aPower = "noPower");
    virtual ~SuperHero();
    virtual void fight() const;
    void print() const override;
    const std::string& getSuperPower() const;
    void setSuperPower(const std::string& thePower);
  private:
    std::string superPower;
};

#endif /* SUPERHERO_H_ */
```

```
/*
 * SuperHero.cpp
 *
 *  Created on: 30.04.2013
 *      Author: rbondere
 */

#include <iostream>
#include <string>
#include "SuperHero.h"
using namespace std;

SuperHero::SuperHero(const string& aName,
                     const string& aPower) :
  ComicCharacter(aName), superPower(aPower)
{
}

SuperHero::~SuperHero()
{
}

void SuperHero::fight() const
{
  cout << getName() << " fights" << endl;
}

void SuperHero::print() const
{
  cout << "Superhero " << getName() << " Superpower: " << superPower << endl;
}
```

```
void SuperHero::setSuperPower(const string& thePower)
{
  superPower = thePower;
}

const string& SuperHero::getSuperPower() const
{
  return superPower;
}
```

```
/*
 * SingingComicCharacter.h
 *
 *  Created on: 30.04.2010
 *      Author: rbondere
 */

#ifndef SINGINGCOMICCHARACTER_H_
#define SINGINGCOMICCHARACTER_H_

#include <string>
#include "ComicCharacter.h"

class SingingComicCharacter: virtual public ComicCharacter
{
  public:
    SingingComicCharacter(const std::string& aName = "");
    virtual ~SingingComicCharacter();
    virtual void dance() const;
    virtual void sing() const;
};

#endif /* SINGINGCOMICCHARACTER_H_ */
```

```
/*
 * SingingComicCharacter.cpp
 *
 *  Created on: 30.04.2010
 *      Author: rbondere
 */

#include <iostream>
#include "SingingComicCharacter.h"
using namespace std;

SingingComicCharacter::SingingComicCharacter(const string& aName) :
  ComicCharacter(aName)
{
}

SingingComicCharacter::~SingingComicCharacter()
{
}

void SingingComicCharacter::dance() const
{
  cout << getName() << " dances" << endl;
}
```

```
void SingingComicCharacter::sing() const
{
  cout << getName() << " sings" << endl;
}
```

```
/*
 * Duck.h
 *
 *  Created on: 25.04.2019
 *      Author: rbondere
 */

#ifndef DUCK_H_
#define DUCK_H_

#include <string>
#include "SingingComicCharacter.h"

class Duck: public SingingComicCharacter
{
  public:
    Duck(const std::string& aName = "");
    virtual ~Duck();
    void print() const override;
    void cackle() const;
};

#endif /* DUCK_H_ */
```

```
/*
 * Duck.cpp
 *
 *  Created on: 30.04.2010
 *      Author: rbondere
 */

#include <iostream>
#include "Duck.h"
using namespace std;

Duck::Duck(const string& aName) :
  SingingComicCharacter(aName)
{
}

Duck::~Duck()
{
}

void Duck::print() const
{
  cout << "Duck " << getName() << endl;
}

void Duck::cackle() const
{
  cout << getName() << " cackles" << endl;
}
```

```
/*
 * DuckHero.h
 *
 *  Created on: 25.04.2019
 *      Author: rbondere
 */

#ifndef DUCKHERO_H_
#define DUCKHERO_H_

#include "Duck.h"
#include "SuperHero.h"

class DuckHero: public Duck, public SuperHero
{
  public:
    DuckHero(const std::string& aName = "",
             const std::string& aPower = "noPower");
    virtual ~DuckHero();
    void print() const override;
};

#endif /* DUCKHERO_H_ */
```

```
/*
 * DuckHero.cpp
 *
 *  Created on: 30.04.2010
 *      Author: rbondere
 */

#include <string>
#include "DuckHero.h"
using namespace std;

DuckHero::DuckHero(const string& aName,
                   const string& aPower) :
  ComicCharacter(aName), // must be here because of virtual inheritance
      Duck(aName), SuperHero(aName, aPower)
{
}

DuckHero::~DuckHero()
{
}

void DuckHero::print() const
{
  Duck::print();
  SuperHero::print();
}
```

```cpp
/*
 * ComicTest.cpp
 *
 *  Created on: 30.04.2013
 *      Author: rbondere
 */
#include <iostream>
#include <typeinfo>
#include "DuckHero.h"

using namespace std;

int main()
{
  DuckHero dh("CrazyDuck", "Flashspeed");
  dh.fight();
  dh.sing();
  dh.print();

  DuckHero* pdh = &dh;
  SuperHero* psh = &dh;
  Duck* pd = &dh;
  ComicCharacter* pcc1 = (Duck*)&dh; // ugly (C-Cast)
  ComicCharacter* pcc2 = dynamic_cast<Duck*> (&dh);
  ComicCharacter* pcc3 = (SuperHero*)&dh; // ugly (C-Cast)
  ComicCharacter* pcc4 = dynamic_cast<SuperHero*> (&dh);
  ComicCharacter* pcc5 = dynamic_cast<SingingComicCharacter*> (&dh);

  cout << hex << showbase;
  cout << "pdh  = " << pdh << endl;
  cout << "psh  = " << psh << endl;
  cout << "pd   = " << pd << endl;
  cout << "pcc1 = " << pcc1 << " (ugly)" << endl;
  cout << "pcc2 = " << pcc2 << endl;
  cout << "pcc3 = " << pcc3 << " (ugly) " << endl;
  cout << "pcc4 = " << pcc4 << endl;
  cout << "pcc5 = " << pcc5 << endl;
  cout << "Type of pdh: " << typeid(*pcc1).name() << endl;
  cout << "Size of dh : " << sizeof(dh);

  return 0;
}
```

13.5 RTTI

```
/*
 * DynCastTest.cpp
 *
 *  Created on: 30.04.2013
 *      Author: rbondere
 */
#include <iostream>
#include <iomanip>
#include <typeinfo>
using namespace std;

class A
{
  public:
    virtual void fooA() {}
    virtual ~A() {}
  private:
    int dataA;
};

class B
{
  public:
    virtual void fooB(){}
    virtual ~B() {}
  private:
    double dataB;
};

class C: public A, public virtual B
{
};

int main()
{
  A* pa = new A;
  B* pb = new B;
  C* pc1 = (C*)pa;
  C* pc2 = dynamic_cast<C*>(pa);
//  C* pc3 = (C*)pb;  // geht nicht, da C von B virtual erbt
  C* pc4 = dynamic_cast<C*>(pb);

  cout << hex << showbase;
  cout << "pa  = " << pa << endl;
  cout << "pb  = " << pb << endl;
  cout << "pc1 = " << pc1 << endl;
  cout << "pc2 = " << pc2 << endl;

  cout << "pc4 = " << pc4 << endl;

  cout << "Typ von pa : " << typeid(*pa).name() << endl;
  cout << "Typ von pb : " << typeid(*pb).name() << endl;
  cout << "Typ von pc1: " << typeid(*pc1).name() << endl;
  delete pa;
  delete pb;
  return 0;
}
```

```
/* Der moegliche Output sieht wie folgt aus:
pa  = 0x80069008
pb  = 0x8006fcc8
pc1 = 0x80069008
pc2 = 0
pc4 = 0
Typ von pa : 1A
Typ von pb : 1B
Typ von pc1: 1A

Bemerkungen:
pc1 und pa haben die selbe Adresse aber unterschiedliche Typen.

pc1 ist unschoen (C-Cast). Es ist zwar eine korrekte Anweisung, ist aber
unsinnig, da ein Pointer nicht auf ein Objekt seiner Oberklasse zeigen
soll (umgekehrt ist ok).

pc2 ist die saubere Variante von pc1. Der Typecast ist erfolgreich, falls pa
wirklich auf ein Objekt der Klasse C oder auf eine UNTER-Klasse von C zeigt.
Da dies nicht der Fall ist, erhaelt pc2 den Nullpointer.

pc3 funktioniert nicht, da B eine VIRTUELLE Basisklasse von C ist. Zudem ist
es wieder ein unschoener C-Cast, der nicht verwendet werden sollte.

pc4 ergibt aus diesem Grund den Nullpointer, ist aber eine korrekte Anweisung.
*/
```