

Zusammenfassung C++ Nick Graf

C++ Allgemein

Gleiche Basisdatentypen wie in C (plain old data types)
Neu Bool mit Werte true und false

Programmierparadigmen:

- prozuderal
- objektorientiert
- generisch (Templates)
- teils funktional (Lambda)

Styleguide

Variablen, Konstanten und Objekte

- mit Kleinbuchstaben beginnen
- erster Buchstaben von zusammengesetzten Wörter gross (mixed case)
- keine Underscores

Bsp. counter, maxSpeed

Funktionen, Klassen, Strukturen, Enums

- mit Grossbuchstaben beginnen
- mixed case
- Name beschreibt Tätigkeit (Funktionen) oder Dinge (Klassen, Strukturen, Enums)

Bsp. getCount(), init(), setMaxSpeed()
MotorController, Queue, Color

Streams <iostream>

- Ein Stream repräsentiert ein sequentieller Datenstrom (z.B Terminal, Tastatur, Dateien)
- Operatoren << (Inserter) und >> (Extractor)
- C++ stellt 4 Standardströme zur Verfügung
 - o cin: Standard-Eingabestrom, normalerweise Tastatur
 - o cout: Standard-Ausgabestrom, normalerweise Bildschirm
 - o cerr: Standard-Fehlerausgabestrom, normalerweise Bildschirm
 - o clog: mit cerr gekoppelt
- Alle Ströme können auch mit einer Datei gekoppelt werden

Eingabe:	Ausgabe (formatiert)
double d;	cout << showbase << hex << 27;
cin >> d;	// Ausgabe: 0x1b

Inline-Funktionen

- inline-Funktionen sollen anstatt defines verwendet werden
- gebrauch bei kleinem Codeumfang und häufiger verwendung (z.B Schleifen)
- Rekursive Funktionen und Funktionen, auf die mit einem Funktionspointer gezeigt wird, werden nicht inlined
- müssen in Headerfiles definiert sein
- Für inlining muss dem Compiler eine Optimierungsstufe mitgegeben werden
clang++ -c -O3 foo.cpp

Vorbelegte Parameter

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

```
// erlaubt sind z.B. die folgenden Aufrufe: // nicht erlaubt sind z.B. diese Deklarationen:
prtDate();                                // 1-3-2009        void prtDate2(int day=7, int month, int year=2009);
prtDate(23);                             // 23-3-2009       void prtDate3(int day, int month=3, int year);
prtDate(15, 6);                          // 15-6-2009
prtDate(24, 7, 2012);                    // 24-7-2012
```

- Parameter können im Funktionsprototypen (nur dort!) Defaultwerte zugewiesen werden
- Bei Funktionsaufruf können Parameter mit Default-Werten weggelassen werden - dann werden die Defaultparameter genommen
- Wenn bei einem Parameter ein Default definiert wird, dann müssen bei allen weiteren Parameter (rechts davon) ebenfalls Defaults definiert werden.
Grund: Parameterübergabe erfolgt in C++ von links nach rechts

Overloading

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

Durch das mehrmalige definieren wird eine Funktion überladen.
Die Identifikation erfolgt über die Signatur, nicht nur über den Namen
Die Signatur besteht aus:

- Name der Funktion plus Parameterliste (Reihenfolge, Anzahl, Typ) (Returntyp wird nicht berücksichtigt)

Die Implementation muss für jede überladene Funktion separat erfolgen.

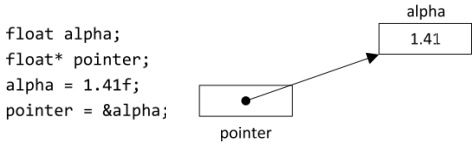
Regeln:

- Haben zwei Deklarationen den selben Rückgabetyt und Parameterliste, wird die zweite als gültige Re-Deklaration der ersten aufgefasst
- Unterscheiden sich zwei Deklarationen nür bezüglich des Rückgabetyts, behandelt der Compiler die zweite als fehlerhafte Re-Deklaration
- Unterscheiden sich beide Deklarationen in Anzahl oder Typ ihrer Parameter, werden sie als zwei verschiedene Deklarationen mit dem selben Funktionsnamen betrachtet (überladene Funktion)

keinesfalls Default-Parameter in überladenen Funktionen verwenden!

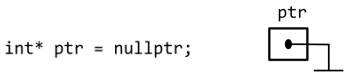
Pointer

Ein Pointer ist eine Variable, welche eine Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann.



C++ Pointer sind typisiert. Der Speicherbereich auf den ein Pointer zeigt wird entsprechend des Pointer-Typs interpretiert. Der Speicherbedarf einer Pointervariable ist so gross, dass die maximale Adresse Platz findet. (z.B 32 Bits für 2^{32} Adressen)

Null-Pointer: Zeigt, dass Pointer auf kein Objekt zeigt. Dem Pointer wird ein definierter Nullwert zugewiesen. Ab C++11 gibt es einen vordefinierten Wert für den Nullpointer: nullptr



Void-Pointer: Einem Void-Pointer kann jeder Pointer zugewiesen werden. Ohne Typecast können nur andere Void-Pointer auf einen Void-Pointer zugewiesen werden. Void-Pointer können nicht dereferenziert werden. In C++ sollten void-Pointer kaum noch angewendet werden.

```
int a;
int* pi = &a;
void* pv = pi;           // ok
double* pd = pv;         // Error (in C erlaubt)
pd = static_cast<double*>pv; // ok, C++-Cast
```

Referenzen

- Alternativname (Alias) für ein Objekt
- können im Gegensatz zu Pointer auch auf Register verweisen
- sind grundsätzlich effizienter als Pointer
- können nach der Definition nicht auf ein anderes Objekt zeigen

```
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)
```

Anwendung:

- Bei Parameterübergabe (call by reference) anstatt Pointer
- Bei Referenz-Rückgabetyt anstatt Pointertyp, d.h als Returntyp
- Generell: Objekte einer Klasse und Strukturvariablen sollen immer by reference übergeben werden (niemals by value)
- Sonst: zurückhaltend einsetzen

! Ein Pointer oder eine Referenz auf eine lokale Variable oder lokales Objekt darf niemals mittels return zurückgegeben werden, da nach Beendigung der Funktion die lokalen Variablen, auf welche verwiesen wird, ungültig sind

call by reference:

1. wenn Parameter in der Funktion verändert werden sollen
2. übergabe von struct/class

Um das Verändern eines Input-Parameters zu verhindern, soll dieser mit const deklariert werden.

Parameterrückgabe und Rückgabe von Objekten by value sind Hauptgründe für langsame C++ Programme

constexpr

- konstanter Ausdruck (constant expression)
- muss vollständig vom Compiler initialisiert werden
- ist bereits zur Compilezeit bekannt

```
constexpr double pi = 3.1415927;
constexpr float x{2.5};
constexpr float z = exp(2.35, 3);
constexpr int i;           // Error, not initialized
int j = 0;
constexpr int k = j + 1;    // Error, j is not a constant expression
```

Gültigkeitsbereiche (Scope)

Lokal (local scope): Ab deklaration bis ende Block

Funktionsprototyp, Funktion: In Funktionskopf deklarierte Bezeichner gelten in der gesamten Funktion

Namensraum (namespace): Ab Deklaration bis Ende Namensraum

Klasse: In einer Klasse deklarierte Bezeichner gelten in der gesamten Klasse

Variablen in inneren Blöcken sind nach aussen nicht sichtbar. Globale Variablen oder Variablen in äusseren Blöcken sind in inneren Blöcken sichtbar. Bei Namensgleichheit ist nur die lokale Variable sichtbar.

Namensräume (Namespaces)

In C und C++ gibt es folgende Namensräume:

- Marken
- Namen von Strukturen, Unions und enums
- Jede Struktur und Union für ihre Feldnamen
- Bezeichner von Variablen, Funktionen typedef-Namen, enum-Konstanten

In C++ können zusätzlich explizit definierte Namensräume verwendet werden. Dadurch können die in C notwendigen Modulkürzel vermieden werden.

Deklaration	using-Deklaration
<pre>namespace myLib1 { int i; void foo(); }</pre>	<pre>int main() { using myLib1::foo; // lokales Synonym foo(); // ruft myLib1::foo() auf }</pre>
using-Direktive	
<pre>namespace myLib2 { int i; void foo(); int goo(); } ... { myLib1::foo(); myLib2::i = 17; }</pre>	<pre>using namespace myLib1; // "importiert" alle Namen aus myLib1 int main() { foo(); // ruft myLib1::foo() auf }</pre>

Namenlose Namespaces

- wird wie ein spezieller Namensraum mit systemweit eindeutigen Namen behandelt
- nur intern verwendete Funktionen und Daten mit namenlosen Namespaces auf den Bereich eingrenzen, in dem die Objekte verwendet werden (guter Programmierstil)

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

using namespace Konflikte

```
using namespace myLib1;
using namespace myLib2;

int main()
{
  foo(); // ist nicht eindeutig, Compiler reklamiert
  myLib2::foo(); // Aufruf muss Namespace vorangestellt werden
}
```

Zugriff auf globale Variablen mit Scope-Operator

```
int zahl = 11; // globale Variable

int main()
{
  int zahl = 22; // lokale Variable
  zahl = zahl + 4; // lokale Variable
  ::zahl = 23; // Zugriff auf globale Variable
}
```

Konzept Datenkapsel (data encapsulation)

- eine Unit besteht aus Funktionen und Daten
- Die Schnittstelle definiert, was für den Nutzer (client) zur Verfügung steht
- auf Daten soll nicht direkt zugegriffen werden können sondern nur über Zugriffsfunktionen
- Datenkapselung kann gut mit Klassen umgesetzt werden

Klassen

Die Klasse ist eine Typendefinition, die Objekte sind die Instanzen (Variablen) der Klasse.

UML-Notation der Klasse

ClassName
-attribute1 : int = 0
-attribute2 : int = 0
+method1()
+method2()

Attribute

Methoden

Sichtbarkeit:

+ public
- private
protected

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

Klasse deklarieren Objekte anlegen

```
class Classname
{
    public:
        ...
    protected:
        ...
    private:
        ...
};
```

```
Stack s;
s.push(23);
Stack* ps = &s;
ps->push(35);
Stack& r = s;
r.push(-56);
```

- public: Elemente können innerhalb und von Aussen angesprochen werden
- fast alle Methoden sind public
 - Attribute sollen nie public sein
- protected: Elemente können innerhalb 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

const

Elementfunktionen, welche den Zustand eines Objektes nicht ändern (Selektoren) sollen explizit mit dem Schlüsselwort const gekennzeichnet werden (Sowol im Prototyp als auch in der Implementierung)

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

! const-Objekte dürfen keine nicht-const Elementfunktionen aufrufen

mutable

Attribut, das auch von const-Elementfunktionen geändert werden kann.

```
class Stack
{
public:
    int pop();
    int peek() const; // peek() ist read-only, liest nur das oberste Element
    bool isEmpty() const;
private:
    int elem[maxElems]; // Array für Speicherung des Stacks
    size_t top; // Arrayindex des nächsten freien Elements
    mutable bool error; // true: Fehler passiert; false: sonst
    // mutable: auch const-Methoden können dieses Attribut setzen
};

int Stack::peek() const
{
    error = top == 0; // 'error' soll auch in const-Methode gesetzt werden können
    if (!error)
        return elem[top-1];
    else
        return elem[top];
}
```

Dynamische Speicherverwaltung

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

! Zugriff auf dynamische Speicherelemente nie verlieren

```
int* pInt = new int; // Speicher für int alloziert
char* pCh1 = new char; // Speicher für char alloziert
char* pCh2 = new char; // Speicher für char alloziert

delete pInt; // Speicher wieder freigeben
delete pCh1;
delete pCh2; // ergibt Fehler, bereits über pCh1 freigegeben

*pInt = 23;
std::cin >> *pCh1;
pCh2 = pCh1;
// pCh2 zeigt nun auch auf die Speicherstelle, auf welche pCh1 zeigt
// Damit geht aber der Zugriff auf die Speicherstelle verloren, auf die
// pCh2 gezeigt hat (memory leak!)
```

Pointer sollten nach delete auf nullptr gesetzt werden Dynamische Allokierung von Arrays

```
delete pInt; // Speicher wieder freigeben
delete pInt; // Speicher ist bereits freigegeben
pInt = nullptr;
delete pInt; // ist problemlos

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

Statt einer Konstanten kann hier auch eine Variable verwendet werden (das ist der Normalfall)

this-Pointer

Der this-Pointer zeigt auf das eigene aktuelle Objekt, welches eine Elementfunktion (Methode) aufgerufen hat. Der this-Pointer wird beim Aufruf jeder Methode implizit als erster Parameter übergeben.

```
const AnyClass& AnyClass::aMethod(const AnyClass& obj)
{
    this->anyFoo(); // Aufruf einer Methode über this
                  // 'this' ist hier unnötig, da Methode implizit mit aktuellem
                  // Objekt ausgeführt wird

    if (this == &obj) // testen, ob eigene Adresse gleich der Adresse von obj ist
        ...

    return *this; // eigenes Objekt zurückgeben
}
```

friend-Elemente

Jede Klasse kann andere Klassen oder Funktionen "zum Freund" erklären. Dadurch werden die Zugriffsregeln durchbrochen. Jeder friend darf auf alle Elemente der Klasse zugreifen, auch auf die privaten.

Konstruktor

- "saubere" Initialisierung des Objekts, d.h alle Attribute müssen auf einen definierten Wert gesetzt werden
- sollte nie explizit aufgerufen werden
- wird vom System implizit aufgerufen, wenn ein Objekt erzeugt wird
- wird auch aufgerufen, wenn durch den new-Operator Speicher angefordert und erhalten wird

Default-Konstruktor

- Konstruktor ohne Parameter
- wird immer aufgerufen, wenn bei der Objekterzeugung keine Parameter mitgegeben werden
- wird vom System automatisch erzeugt, wenn nicht explizit definiert

```
class TString
{
public:
    TString();
    size_t getLen() const;
private:
    size_t len;
    char* str;
};
// mit Anweisungen
TString::TString()
{
    len = 0;
    str = nullptr;
}
```

```
// oder: mit Initialisierungsliste (ist besser)
TString::TString()
: len(0), str(nullptr) // bis C++11: len(0), str(nullptr)
```

Überladen von Konstruktoren

```
class TString
{
public:
    TString(); // Default-Constructor
    TString(const char* p);
    size_t getLen() const;
private:
    size_t len;
    char* str;
};
TString::TString(const char* p)
{
    if (p == nullptr)
    {
        len = 0;
        str = nullptr;
    }
    else
    {
        len = strlen(p);
        str = new char[len+1];
        memcpy(str, p, len+1);
    }
}
```

Copy-Konstruktor

- wird verwendet um Objekte zu kopieren
- erhält als Parameter immer eine konstante Referenz auf ein Objekt der Klasse

```
TString(const TString& s); // Copy-Constructor
```

```
// welche Konstruktoren werden aufgerufen?
TString str1("Hello World"); // normaler Constructor TString(const char* p)
TString str2 = str1; // Copy-Constructor
TString str3{str1}; // Copy-Constructor
```

Ist Initialisierung von str2, nicht Zuweisung!

Copy-Konstruktor wird automatisch aufgerufen wenn...

- ein Objekt erzeugt und 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 Referenzrückgabe)

Explizite Konstruktoren

Durch die Kennzeichnung explicit kann der Ctor nur explizit aufgerufen werden

```
explicit TString::TString(int number);
```

```
TString str1 = TString(12345); // ok (explicit)
TString str2 = 12345; // nicht erlaubt (implicit call)
str2 = 78; // nicht erlaubt (implicit call)
str1 = 567; // nicht erlaubt (implicit call)
```

Member In-Class Initialization

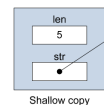
Ab C++11 können bei der Klassendeklaration den Attributen Initialisierungswerte zugewiesen werden.

-> Äquivalent zur Initialisierungsliste bei Ctors

```
class TString
{
public:
    TString();
    ...
private:
    size_t len = 0;
    char* str = nullptr;
};
```

Shallow- vs. Deep Copy

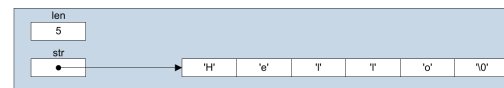
! Wenn ein Objekt Speicher auf dem Heap alloziert, muss ein eigener Copy-Constructor definiert werden (in allen anderen Fällen meist nicht)



Shallow Copy: Pointer welche auf Heap zeigen, kopieren nur Adresse aber nicht Speicher

Deep Copy: Auch der Speicherbereich wird kopiert. Deep Copy muss in einem selbst definiertem Copy-Constructor implementiert werden

```
TString::TString(const TString& s)
: len(s.len)
{
    if (s.str == nullptr)
    {
        str = nullptr;
    }
    else
    {
        str = new char[len+1];
        memcpy(str, s.str, len+1);
    }
}
```



Destruktor

- vollständige "Zerstörung" eines nicht mehr benötigten Objekts
- häufigste Aufgabe ist Freigabe von nicht mehr benötigtem Speicher auf dem Heap
- sehr häufig (wenn kein Speicher auf dem Heap vorhanden ist) wird kein Destruktor definiert
- werden automatisch aufgerufen, wenn der Gültigkeitsbereich des definierten Objekts ausläuft
- Reihenfolge des Aufrufs der Destrukturen ist umgekehrt wie die der Konstruktoren (das zuletzt erzeugte Objekt wird zuerst aufgeräumt)
- nicht definierte Destrukturen werden automatisch erzeugt

```
class TString
{
public:
    TString(); // Default-Constructor
    TString(const TString& s); // Copy-Constructor
    TString(const char* p);
    explicit TString(int number);
    ~TString(); // Destruktor
    size_t getLen() const;
private:
    size_t len;
    char* str;
};
TString::~TString()
{
    delete[] str; // weil str ein Array auf dem Heap ist
}
```

Kanonische Form von Klassen

Klasse kann wie ein "normalen" Datentyp (POD verwendet werden. Dazu müssen drei Bedingungen erfüllt sein:

1. ein korrekter Default-Konstruktor, plus evtl. weitere Konstruktoren
2. ist die Klasse dynamisch braucht es einen Zuweisungsoperator und ein Copy-Konstruktor
3. ein (virtueller) Destruktor garantiert die korrekte Zerstörung von Objekten

Typumwandlung (Type Cast)

Unsafe conversion:

- signifikante Stellen können bei Umwandlung verloren gehen
- typisch bei Umwandlung von "grösseren" in "kleineren" Typ, z.B von double nach int
- > bei int ist sowohl die Genauigkeit kleiner als auch die maximal darstellbare Zahl

Safe conversion:

- keine signifikanten Stellen können bei Umwandlung verloren gehen

Implizite Typumwandlung:

- wird auch als Standard-Typumwandlung bezeichnet
- braucht kein expliziten Cast

Explizite Typumwandlung:

- C++ kann mit Hilfe von 6 verschiedenen cast-Operatoren eine explizite Typumwandlung bewirken
- der Programmierer gibt explizit an was er will, übernimmt aber auch die Verantwortung

1&2. C-Stil und Funktionsstil

(Zieltyp)expression

```
int a = (int)4.6; // a == 4
```

Zieltyp(expression)

```
int a = int(4.6); // a == 4
```

3. const_cast

Ausschliesslich vorübergehende Entfernung des const-Qualifikators

```
const_cast<Zieltyp>(expression)
```

5. dynamic_cast

Umwandlung von polymorphen Objekten

4. static_cast

Umwandlung von Objekten einer Klasse auf Objekte einer Basisklasse oder Umwandlung mittels einer Umwandlungs-Funktion

```
static_cast<Zieltyp>(expression)
```

```
uint16_t word = 0x2AFE;
uint8_t byte = static_cast<uint8_t>(word);
```

6. reinterpret_cast

neue Interpretation des zugrundeliegenden Bitmusters
keine Prüfung der Typkompatibilität
Ergebnis ist Plattformspezifisch d.h nicht portabel

```
reinterpret_cast<Zieltyp>(expression)
```

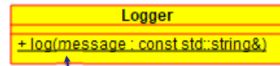
```
char* p = new char[20];
int* pint = reinterpret_cast<int*>(p);
```

static

Wenn ein Attribut mit static gekennzeichnet wird, teilen sich alle Objekte dieser Klasse eine einzige Instanz des Attributs. Es ist nur einmal im Speicher vorhanden.

```
class Logger
{
public:
    static void log(const std::string& message);
};

int main()
{
    // Benutze log aus Logger, ohne Instanz:
    Logger::log("C++ ist super!");
    return 0;
}
```



Darstellung im UML-Klassendiagramm: unterstrichen

```
class T
{
...
// static int nrOfObjects = 34; // Deklaration
static int nrOfObjects; // Deklaration
};
```

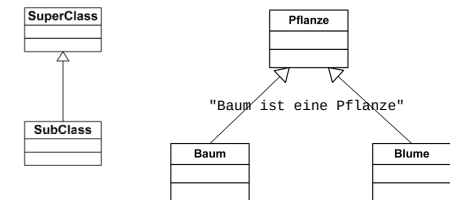
Die Initialisierung muss in der Definition (.cpp) erfolgen. Das Schlüsselwort *static* muss hier weggelassen werden.

```
static int T::nrOfObjects; // Definition (ist nötig)
int T::nrOfObjects = 34; // Definition (ist nötig)
```

```
T myT;
myT.nrOfObjects++; // Zugriff über Objekt (falls public)
T::nrOfObjects++; // Zugriff über Klasse (falls public)
```

- static deklarierte Methoden können nur auf static deklarierte Attribute zugreifen
- static Attribute müssen irgendwo in der .cpp-Datei definiert werden (zusätzlich zur Deklaration in .h-Datei)
- static Methoden und Attribute benötigt man selten
- bei der OOP sind static deklarierte Attribute globalen Variablen und Funktionen vorzuziehen

Vererbung



- Generalisierung/Spezialisierung
- SubClass erbt sämtliche Eigenschaften von SuperClass
- ist eine Beziehung (SupClass ist eine SuperClass)
- Pfeilspitze ist ein geschlossenes Dreieck
- Ein Objekt der vererbten Klasse enthält alle Teile der Basisklasse
- Das Objekt der Unterklasse ist mindestens so groß wie die Basisklasse

```
class SubClass : public SuperClass
{
public:
...
protected:
...
private:
...
};
```

public ist Normalfall (private und protected sind auch möglich)

Inheritance	In base class	In derived class
public (normal case)	public protected private	public protected private
protected	public protected private	public protected private
private	public protected private	public protected private

```
class Fahrzeug {
public:
    int kmStand;
protected:
    int vorbesitzer;
private:
    int id;
};

class Auto : public Fahrzeug {
    // kmStand: public
    // vorbesitzer: protected
    // id: unsichtbar in Klasse Auto!
};

class Auto : protected Fahrzeug {
    // kmStand: protected
    // vorbesitzer: protected
    // id: unsichtbar in Klasse Auto!
};

class Auto : private Fahrzeug {
    // kmStand: private
    // vorbesitzer: private
    // id: unsichtbar in Klasse Auto!
};
```

Chaining bei Ctors und Dtors

- der Ctor einer Klasse initialisiert explizit nur die eigenen Attribute
- für die Initialisierung der Basisklasse wird der Ctor der Basisklasse verwendet
- Wichtig: die Elemente der Basisklasse müssen immer als erstes initialisiert werden

```
Book::Book(const string& aName, const string& aTitle, unsigned int aPartNr,
            unsigned int aRating, const string& aComment, double aPrice, unsigned int aYear,
            double aWeight, const string& aAuthor,
            const string& aIsbn, const string& aPublisher, unsigned int aEdition) :
    Article{aName, aTitle, aPartNr, aRating, aComment, aPrice, aYear, aWeight},
    author{aAuthor}, isbn{aIsbn}, publisher{aPublisher}, edition{aEdition} // eigene Attribute
{
}
```

- Destruktoren werden auch 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 Dtor der Basisklasse wird am Schluss des Dtors immer automatisch aufgerufen

Wird ein Objekt der Unterklasse erstellt, muss zuerst der Ctor der Oberklasse aufgerufen werden, bevor der Ctor der Unterklasse aufgerufen wird. Dies passiert ggf. automatisch mit dem Default-Ctor der Oberklasse. Es ist aber meistens nützlich, in der Initialisierungsliste explizit einen geeigneten Ctor der Oberklasse aufzurufen.

Dasselbe gilt in umgekehrter Reihenfolge für die Destruktoren: Zuerst wird der Dtor der Unterklasse ausgeführt, danach - ebenfalls automatisch - alle Destruktoren der Oberklassen in aufsteigender Reihenfolge.

Substitutionsprinzip

- Objekt einer Oberklasse kann Objekte einer beliebigen Unterklasse aufnehmen
- Objekt einer Unterklasse kann keine Objekte der Oberklasse aufnehmen

```
class SuperClass {};
```

```
class SubClass : public SuperClass {};
```

```
SuperClass super;
SubClass sub;
super = sub; // ok
sub = super; // geht nicht
```

Aufruf des Copy-Constructors von ComicCharacter

```
void fooVal(ComicCharacter o)
{
    o.print();
}
```

hier wird nur der ComicCharacter-Teil von SuperHero s geschrieben

```
ComicCharacter c;
SuperHero s;

fooVal(c);
fooVal(s); // Substitutionsprinzip
```

```
void fooRef(const ComicCharacter& o)
{
    o.print();
}
```

hier wird der ganze SuperHero geschrieben (dank des Referenzparameters)

```
ComicCharacter c;
SuperHero s;

fooRef(c);
fooRef(s); // Substitutionsprinzip
```

Polymorphismus (Dynamic Binding)

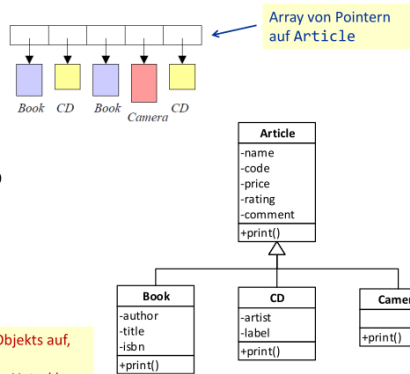
- Eine Klasse, welche min. eine virtuelle Funktion deklariert, heisst virtuell (polymorph)
- Elementfunktionen, die dynamisch gebunden werden, muss bei der Deklaration das Schlüsselwort `virtual` vorangestellt werden. Sie sind dann automatisch auch in den Unterklassen virtuell.
- In der Abgeleiteten Klasse muss die Funktion mit `override` gekennzeichnet werden, falls sie überschrieben wird.
- Kommen in einer Klasse virtuelle Elementfunktionen vor, muss auch der Destruktor als `virtual` gekennzeichnet werden
- Der Destruktor muss auch dann virtuell sein, wenn ein Objekt einer Unterklasse dynamisch erzeugt wird und einem Pointer auf die Basisklasse zugewiesen wird (Supstitutionsprinzip)
- Konstruktoren sind nie virtuell!
- nicht mit Funktionsüberladung verwechseln!

Beispiel: Webshop

```
Article* a[maxArticle];
a[0] = new Book;
a[1] = new CD;
a[2] = new Book;
...
```

```
for (size_t i = 0; i < maxArticle; ++i)
{
    if(a[i] != nullptr)
    {
        a[i]->print();
    }
}
```

*a[i]->print() ruft die print-Methode des Objekts auf, auf das a[i] gerade zeigt.
Diese Schleife ändert sich nicht, wenn eine neue Unterklasse hinzugefügt wird!*



- o Static binding (early binding) -> Normalfall
 - zur Compilezeit wird festgelegt, welcher (Elementfunktions-) Code ausgeführt wird
 - Auswahl der richtigen Funktion erfolgt bereits beim Übersetzen, d.h ohne Overhead
- o Dynamic binding (late binding)
 - Bei dynamischen Aufrufen (über Pointer oder Referenzen) erfolgt die Auswahl der richtigen Funktion erst zur Laufzeit. Dies ist mit etwas Overhead verbunden

Statischer Aufruf

```
Duck donald;
SuperHero luckyLuke;
```

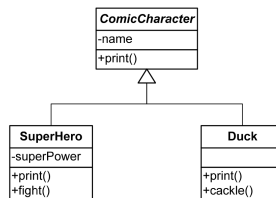
```
// direkter (statischer) Aufruf über die
// entsprechenden Objekte
// Auflösung durch den Compiler
donald.print(); // Duck::print()
luckyLuke.print(); // SuperHero::print()
```

dynamischer Aufruf

```
void printCC1(const ComicCharacter& c)
{
    c.print(); // dynamische Auflösung
}

void printCC2(const ComicCharacter* pc)
{
    pc->print(); // dynamische Auflösung
}

Duck donald;
SuperHero luckyLuke;
// Auflösung erst durch das Laufzeitsystem
printCC1(donald); // Duck::print()
printCC2(&luckyLuke); // SuperHero::print()
```



final specifier

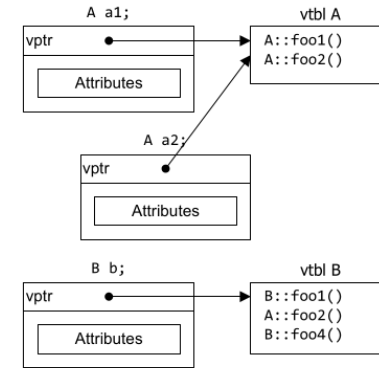
Elementfunktion: diese Funktion kann in einer Unterklasse nicht mehr überschrieben werden
Klasse: von dieser Klasse kann nicht mehr geerbt werden

Virtual Function Table (vtbl)

Das System vermerkt in der vtbl der Reihe nach die Adressen der für eine Klasse gültigen virtuellen Elementfunktionen. Für jede polymorphe Klasse wird eine vtbl angelegt. Jedes Objekt einer polymorphen Klasse enthält einen Virtual Table 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:
    void foo1() override;
    virtual void foo4();
};
```



Abstrakte Klassen

- Eine Methode soll zwar da sein, aber erst in einer der Unterklassen festgelegt werden
- Solche Methoden heissen abstrakt
- Besitzt eine Klasse min. eine geerbte oder eigene abstrakte Methode, handelt es sich um eine Abstrakte Klasse
- Eine Abstrakte Klasse ist unvollständig, von ihr können keine Objekte gebildet werden
- alle abstrakten Methoden müssen in einer Unterklasse durch overriding konkretisiert werden, damit Objekte gebildet werden können
- Im UML-Klassendiagramm werden die Namen von abstrakte Klassen und Methoden kursiv geschrieben

virtual double getArea() = 0;

Mehrfachvererbung

```
class SingingWaiter : public Waiter, public Singer
```

Um zu vermeiden, dass bei Mehrfachvererbung Attribute einer gemeinsamen Basisklasse mehrfach im Speicher vorkommt, kann virtuel geerbt werden.

```
class SingingComicCharacter : virtual public ComicCharacter
{
    ...
};

class SuperHero : virtual public ComicCharacter
{
    ...
};
```

Run-Time Type Information, RTTI

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

Versucht, den Zeiger p in einen Zeiger auf ein Objekt des Typs SuperHero umzuwandeln. Die Umwandlung wird durchgeführt, wenn p tatsächlich auf ein Objekt vom Typ SuperHero zeigt, andernfalls ist das Resultat der Umwandlung der Nullpointer.

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

Ergibt eine Referenz auf ein Objekt des Typs std::type_info. Diese Klasse beinhaltet u.a eine Methode name(), welche den Namen der Klasse zurückgibt.

!RTTI muss und soll nur äusserst selten eingesetzt werden

Operator overloading

Überladbare Operatorfunktionen in C++

new	delete	new[]	delete[]		Zwingend als Elementfunktion zu implementieren
+	-	*	/	%	
^	&		~	!	Zuweisungs- =
=	<	>	+=	-=	
*=	/=	%=	^=	&=	Index- []
/=	<<	>>	>>=	<<=	
==	!=	<=	>=	&&	Funktionsaufruf- () und
//	++	--	,	->*	Zeigeroperator ->
->	()	[]			

- Anzahl der Operanden (Argumente) muss gleich sein wie beim ursprünglichen Operator
- Priorität des überladenen Operators kann nicht ändern
- Neue Operatoren können nicht eingeführt werden
- Default-Argumente sind bei Operatoren nicht möglich
- Nicht gefordert aber sinnvoll: die neue Operation sollte intuitiv dem bestehenden Operator zugeordnet werden können

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

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

Nur 1 Argument, das zweite ist das aktuelle Objekt (*this)

```
TString s1{"Hallo"};
TString s2{"Hello"};
if (s1 < s2) // s1 < s2 entspricht s1.operator <(s2)
{ ... }

if (s2 < "hoo") // entspricht s2.operator <("hoo")
{ ... }

if ("goo" < s2) // entspricht "goo".operator <(s2)
{ ... }

if (TString{"goo"} < s2) // waere ok
{ ... }
```

"hoo" wird dabei in ein TString umgewandelt

Geht nicht, da der Typ von "goo" ein char* ist. Der Typ des linken Operanden muss zwingend ein TString sein.

Operator overloading als "normale" Funktion

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

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

2 Argumente, da der Operator ausserhalb einer Klasse definiert ist, d.h. *this gibt es nicht

```
// Elementfunktion der Klasse TString
bool TString::operator <(const TString& s) const
{
if (s.len == 0)
return false;
else
return (strcmp(str, s.str) < 0);
}

// "normale Funktion"
bool operator <(const TString& s1, const TString& s2)
{
if (s2.len == 0)
return false;
else
return (strcmp(s1.str, s2.str) < 0);
}
```

Templates

Single Source-Prinzip: ein bestimmtes Problem (Algorithmus) wird nur einmal unabhängig vom Typ gelöst

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.

```
template<typename ElemType>
ElemType minimum(ElemType elemField[], size_t fieldSize);

template<typename A, typename B>
int foo(A a, B b, int i);
```

```
template<typename ElemType>
ElemType minimum(const ElemType elemField[], size_t size);

template<typename ElemType>
ElemType minimum(const ElemType elemField[], size_t size)
{
size_t min = 0; // Index des kleinsten Elementes
for (size_t i=1; i < size; ++i)
{
if (elemField[i] < elemField[min])
min = i;
}
return elemField[min];
}
```

Der Operator < muss für jeden verwendeten Typ definiert sein

inline bei Templates

```
template<typename Any>
inline void swapIt(Any& a, Any& b)
{
Any temp; // temp: a variable of type Any
temp = a;
a = b;
b = temp;
}
```

Ausprägung

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

Explizite Qualifizierung

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

Klassen-Template

Deklaration

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

Definition

```
template<typename ElemType, size_t size>
void Stack<ElemType, size>::push(const ElemType& elem)
{
// implementation goes here
}
```

Nutzung (Ausprägung)

```
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
```

Explizite Ausprägung

```
template<typename KeyType, typename ElemType>
class Map
{
...
}

class Map<int, TString>;
// Der Compiler prägt nun diese Kombination explizit aus
```

Da ein Template nur eine Codeschablone ist, muss sowohl die Deklaration als auch die Definition inkludiert werden. Der Compiler muss die Definition des Templates zwingend kennen.

Die Definition des Templates muss deshalb ebenfalls im Headerfile gemacht werden.

Exceptions

Error (Fehler): Abweichung zur Spezifikation ("falsch implementiert")
Errors sollten bei der Verifikation (Testing) entdeckt und eliminiert werden.

Exception (Ausnahme): abnormale (aber vorhersehbare und mögliche) Bedingung bei der Programmausführung

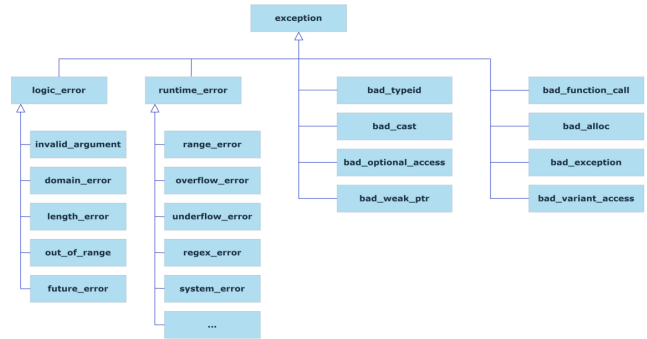
Exceptions werden in Form eines Objekts am Ort ihres Auftretens geworfen (explizit oder auch "automatisch") (to throw)

Exception Handler versuchen, diese Exception-Objekte zu fangen (to catch)

```
try
{
    // Code, der eine Exception auswerfen könnte
}
catch (const MyExceptionClass& exc)
{
    // wenn ein Objekt der Klasse MyExceptionClass oder einer Unterklasse
    // davon geworfen wurde, dann kann dieser Handler das Objekt fangen
}

// u.U. weitere Catches
```

Exception-Hierarchie in C++



Exception	Header
exception	<exception>
bad_exception	<exception>
logic_error	<stdexcept>
invalid_argument	<stdexcept>
domain_error	<stdexcept>
length_error	<stdexcept>
out_of_range	<stdexcept>
future_error	<future>
runtime_error	<stdexcept>
range_error	<stdexcept>
overflow_error	<stdexcept>
underflow_error	<stdexcept>
regex_error	<regex>
system_error	<system_error>
ios_base::failure	<ios>
bad_typeid	<typeinfo>
bad_cast	<typeinfo>
bad_optional_access	<optional>
bad_weak_ptr	<memory>
bad_function_call	<functional>
bad_alloc	<new>
bad_variant_access	<variant>

- ein oder mehrere Exception Handler können hintereinander definiert werden
 - einzelne catch-Handler müssen sich in den Parametern unterscheiden
 - kommt eine Exception geflogen, wird der erste passende Handler genommen
- Ein passender Handler macht ein catch auf genau diese Exception oder auf eine Unterklasse derselben

Der allgemeinste Handler (am meisten oben in der Hierarchie) muss der letzte catch-Handler sein (weil er alles fangen kann)



throw runtime_error();



catch(const logic_error& e){}



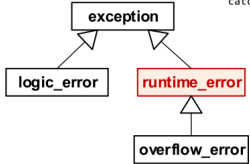
catch(const overflow_error& e){}



catch(const runtime_error& e){}



catch(const exception& e){}



```
graph TD
    exception --> logic_error
    exception --> runtime_error
    runtime_error --> overflow_error
```

Catch all

Der folgende Handler fängt ausnahmslos alle Exceptions ab (und muss wenn gewünscht deshalb immer als letzter aufgeführt werden):

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

Auslösen (Werfen) von Ausnahmen

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

```
class Xcpt
{
public:
    Xcpt(const char* text);
    ~Xcpt();
    const char* getDiagStr() const;
private:
    const char* diagStr;
};

void allocateFoo()
{
    b1();
    if (nullptr == allocation())
        throw Xcpt{"Allocation failed!"};
    b2();
}
```

```
// Testprogramm
void testFoo()
{
    a1();
    try
    {
        a2();
        allocateFoo();
        a3();
    }
    catch (const Xcpt& exc)
    {
        cout << "Caught exception. Text: "
              << exc.getDiagStr() << endl;
    }
    a4();
}
```

Kein passenden Exception Handler

- 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 `std::terminate()` aufgerufen.
- `std::terminate()` ruft wiederum den aktuell registrierten `std::terminate_handler` (dies ist ein Pointer auf eine beliebige Funktion)
- Per Default ist bei `std::terminate_handler` ein Pointer auf die Funktion `std::abort()` hinterlegt. Diese beendet das Programm, ohne dass etwaige Destruktoren noch bestehender Instanzen ausgeführt werden!
- Falls man ein anderes Verhalten wünscht, kann man mittels `set_terminate(std::terminate_handler f);` einen selbst definierten `std::terminate_handler` registrieren.

Exception Propagation

Innerhalb eines Exception Handlers kann eine Exception mittels **throw;** weitergereicht werden

Die Exception wird dann nicht etwa an das nächste **catch()** weitergeleitet, sondern an die aufrufende Funktion

Anhang

Deklarationsreihenfolge in der Headerdatei (*.h)

1. Dateikommentar
2. `#include` der verwendeten System-Header (iostream, etc.)
`#include <...>`
3. `#include` der projektbezogenen Header (`#include "..."`)
4. Konstantendefinitionen
5. typedefs 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!

Punkte 2-7 sind innerhalb
des include-Guards

Reihenfolge in der Implementierungsdatei (*.cpp)

1. Dateikommentar
2. `#include` der *eigenen* Headerdatei (z.B. `myUnit.h`)
3. `#include` der verwendeten System-Header (iostream, etc.)
4. `#include` der projektbezogenen Header
5. allenfalls globale Variablen und statische Variablen
6. Präprozessor-Direktiven
7. Funktionsprototypen von lokalen, internen Funktionen (in nameless Namespace)
8. Definition von Funktionen und Klassen
(Kommentare aus Headerdatei nicht wiederholen!!)

makefile

```
# makefile
cc=clang
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog

$(EXE): $(OBJ)
$(cc) $(LFLAGS) -o $@ $(OBJ)

foo.o: foo.h goo.h foo.c
$(cc) $(CFLAGS) -o $@ foo.c

goo.o: goo.h goo.c
$(cc) $(CFLAGS) -o $@ goo.c

main.o: foo.h main.c
$(cc) $(CFLAGS) -o $@ main.c

clean:
rm -f $(OBJ) $(EXE)
```

Targets: Diese können mit make angesprungen werden,
z.B. `make foo.o`
Wenn make ohne Parameter aufgerufen wird, dann wird
das erste Target angesprungen.

Abhängigkeitsliste, d.h. von diesen
Dateien ist das Target abhängig

Befehl, der ausgeführt werden muss, falls
etwas geändert hat.
Wichtig: erstes Zeichen muss ein TAB sein.