Prog C++

FS 2026 – Prof. Dr. Christian Werner

Autoren: Fabian Steiner
Mitwirkende: Mirko Ratti

V2.0.20260216 https://github.com/Iceteavanill/Cheatsheet-Prog_Cpp_FS_2024_OST

1 Basics

1.1 Grundsätzlich

C++ ist sehr ähnlich zu C. Es gibt aber einige wichtige Änderungen zu C:

- zu benutzender Compiler heisst jetzt `clang++`
- Nativen Bool Typ : **bool**
- richtiges Konstanten Schlüsselwort : **constexpr**
- Standartbibliotheken haben kein .h mehr beim Aufrufen
- C Standartbibliotheken können weiter verwendet werden, haben aber ein C im Header(stdio.h → cstdio)
- Char-Literale (z. B. 'A') haben in C++ den Datentyp char (in C war es int)(Brauchen ein const)
- Es gibt benannte Namensräume. Wichtigster Namensraum für die STL: `std`

1.2 Hello world C++

```cpp
#include <iostream> // iostream fuer Kommandozeilenausgabe
using namespace std; // dadurch entfaellt das "std::" vor cout
int main() // 'void' can be omitted
{
    cout << "Hello World!" << endl; // "Einschieben" von hello
      world in den stream
    return 0; // ein return 0 ist in c++ am Ende der main
     funktion fakultativ
}
```

1.3 Range-based for Loop

Range based for Loops laufen automatisch ein Array ab, ohne das zuerst die Grösse ermittelt werden muss. Die C for loop syntax wurde etwas erweitert. Syntax:

```cpp
for(<DataType> <IteratorName> : arrayName){
    ;//Loop where <IteratorName> is each element
}
```

Siehe: Als Array Name wird ein `Pointer` auf das Array verlangt, nicht auf das erste Element! Ein foreach funktioniert **nicht** mit einem **Pointerpointer**.

1.4 Streams

Ein Stream repräsentiert einen generischen sequentiellen Datenstrom. Z.b: ein Eingabefeld, Dateien oder Netzwerktraffic.
Die wichtigsten Operatoren sind:

- << : Inserter → Daten einfügen
- >> : Extractor → Daten herausholen

Für Standardklassen sind diese Operatoren bereits definiert.

Standardstreams

- **cin** : Standard Eingabe
- **cout** : Standard Ausgabe
- **cerr** : Standard Fehlerausgabe,
- **clog** : Standard Log Ausgabe, möglicherweise mit cerr gekoppelt

`cin`/`cout` Immer ganz links!, operatoren klar und lesbar anordnen

1.4.1 Streamformatierung

Formatierungen der Streams können mit folgenden Schlüsselwörtern erreicht werden:

Flag	Wirkung
boolalpha	bool-Werte werden textuell ausgegeben
dec	Ausgabe erfolgt dezimal
fixed	Gleitkommazahlen im Fixpunktformat
hex	Ausgabe erfolgt hexadezimal
internal	Ausgabe innerhalb Feld
left	linksbündig
oct	Ausgabe erfolgt oktal
right	rechtsbündig
scientific	Gleitkommazahl wissenschaftlich
showbase	Zahlenbasis wird gezeigt
showpoint	Dezimalpunkt wird immer ausgegeben
showpos	Vorzeichen bei positiven Zahlen anzeigen
skipws	Führende Whitespaces nicht anzeigen
unitbuf	Leert Buffer des Outputstreams nach Schreiben
uppercase	Alle Kleinbuchstaben in Grossbuchstaben wandeln

Folgende Funktionen setzten Header <**iomanip**> voraus:

Manipulator	Effekt / Funktionsweise	Gültigkeit
`setw(n)`	Setzt die minimale Feldbreite. Ist der Inhalt kürzer, wird mit Füllzeichen (Standard: Leerzeichen) aufgefüllt.	**Einmalig. Auch für Strings.**
`setprecision(n)`	**Ohne `fixed`**: legt die Gesamtzahl signifikanter Stellen fest. **Mit `fixed`**: legt nur die Anzahl der Nachkommastellen fest.	**Dauerhaft (für den Stream).**
`fixed`	Erzwingt Festkommadarstellung und deaktiviert die wissenschaftliche Notation. Reihenfolge zu `setprecision` ist egal.	**Dauerhaft.**

1.5 Namespaces

Namespaces helfen, Namenskonflikte zu verhindern. Sie fügen ein Namensvorsatz zu allen Variablen. Können sehr mächtig sein, führt aber zu Komplexität
Besser mit kleine Buchstaben starten

```cpp
void f(); // name in global Namespace (1. f())
namespace A{void f();} // namespace A (2. f())
f(); // f von global Namespace
A::f(); // f von namespace A
using namespace A;
::f(); // 1. f()
f(); // 2. f()
```

2 Objektorientierung

Objektorientierung soll es erleichtern, eine Abbildung der Realität zu erstellen. Viele Dinge aus der Wirklichkeit können als Modell dargestellt / simplifiziert werden. Diese Modelle können in Software in sogenannte Objekte `umgewandelt` werden.

2.1 Objekte

Objekte stellen Dinge, Sachverhalte oder Prozesse dar. Sie sind ein rein gedankliches Konzept.
Sie kennzeichnen sich durch:

- eine **Identität**, welche es erlaubt Objekte voneinander zu unterscheiden
- **statische Eigenschaften** zur Darstellung des **Zustandes** des Objekts in Form von **Attributen**
- **dynamische Eigenschaften** zur Darstellung des **Verhaltens** des Objekts in Form von **Methoden**

2.2 Klassen & Instanzen

Ähnliche Objekte können in **Klassen** zusammengefasst werden, um Programmieraufwand tiefer zu halten. Verwendete Objekte werden als **Instanz** eingesetzt. Diese Objekte haben dieselben Attribute, allerdings andere Werte.
Eine Beispielklasse:

```cpp
#include <string>//Stl Strings
class Student {
    public: // folgende Elemente sind Public
        int IDNumber;
        void doStuff(int time = 0);
    private: // folgende Elemente sind Private(standart)
        std::string name;//String
        float bierkonsum;
        unsigned long long int investedHoursInET;
};
// ------ start cpp ------
//implementierung der doStuff Funktion
void Student::doStuff(int time = 0) {
} //ToDo

int main(){
    Student typETStudent;//instanz erzeugen
    typETStudent.IDNumber = 1;//Attribute veraendern
    typETStudent.doStuff();//Methode rufen
}
```

2.2.1 Header-Datei

Reihenfolge in der Header-Datei:

1. Dateikommentar mit Lizenzvereinbarung.
2. *`Includes` des verwendete System Header*
3. *`Includes` der Projektbezogenen Header*
4. *Definition von Konstanten*
5. *Typedef's und Definitionen von Strukturen*
6. *allenfalls extern-Deklaration von Global-Variablen*
7. *Funktionsprototypen, ink. Kommentar der Schnittstelle, bzw. Klassendeklarationen*

Die *blauen* Punkte müssen sich im Includeguard befinden(#pragma once). Pro Header-Datei sollte nur eine Oberklasse deklariert sein. Kleine Unterklassen können im gleichen Header stehen.

2.2.2 Reihenfolge der Implementation

1. Dateikommentar mit Lizenzvereinbarung
2. `includes` der eigenen Header
3. `includes` der Projektbezogenen Header
4. `includes` der verwendeten System Header
5. allenfalls globale und statische Variablen
6. Präprozessor-Direktiven
7. Funktionsprototypen von lokalen, internen Funktionen (in nameless Namespace)
8. Definition von Funktionen und Klassen

2.3 UML

Die `Unified Modeling Language` ist eine normierte Sprache um Objekte grafisch darzustellen. Sie geht extrem ins Detail. Darum hier nur ein Ausschnitt:

2.3.1 UML-Klassendiagramm Notation

Besteht immer aus 3 Boxen:

Student
+ IDNumber : int # name : std::string - bierkonsum : float # investedTimeInET : long int
- doStuff(time : int&) : void + *useless(void)* : retType*

Zu oberst der Name, in der Mitte Attribute(Variablen) und unten Methoden. Die Methoden und Attribute fangen mit einem Symbol an, welches die Sichtbarkeit definiert und haben ein bestimmtes Merkmal:

- \+ : public (überall sichtbar)
- \- : private (nur in aktueller Klasse sichtbar)
- # : protected (in aktueller und Unterklassen sichtbar)
- *Italic* : Funktion oder Klasse ist Abstrakt (=0)
- Unterstrichen : Funktion oder Attribut ist static
- `<Name der Klasse>` : Konstruktor (ohne <>)
- `~<Name der Klasse>` : Destruktor (ohne <>)

2.4 class vs struct

Eine class und struct können fast identisch verwendet werden. Eine class hat standardmässig Sichtbarkeit private, Struct public(wenn nichts definiert wird).

2.5 Inkludierte Dokumentation

Mann kann im H-File direkt die Dokumentation zu der Schnittstelle schreiben. Das ermöglicht das einfachere Verwenden der Schnittstelle. Dies wird in einem Blockkommentar, nach folgendem Muster realisiert:

```
/**
 * @brief Kurzzusammefassung der Klasse
 *
 */
class stuff{
public:
    /**
     * @brief Funktion von x
     *
     */
    double x;
    /**
     *
     * @brief Kurzzusammefassung der Funktion
     *
     * @param eingang Funktion des Parameter
     * @return nix, aber falls...
     *
     */
    void doStuff(stuff eingang);
};
```

Es kann immer mehr geschrieben werden, man sollte sich aber auf das Nötige beschränken.

3 Konstruktoren und Destruktoren

Konstruktoren und Destruktoren sind spezielle Methoden von Klassen. Diese haben immer denselben Namen wie die Klasse und **kein** Rückgabetyp(auch nicht void). Aufrufparameter haben folgende Bedeutung:

- **`Keine`** : Default Konstruktor
- **`const-Referenz auf eigene Klasse`** : copy Konstruktor
- **Alles andere** : User-Defined

3.1 Konstruktoren

Konstruktoren bereiten die Instanz auf ihre Funktion vor.

Wichtig ist, dass ein Konstruktor immer **alle** Attribute der Klasse initialisiert. Konstruktoren werden wie folgt aufgerufen:

3.1.1 User-Defined

- Default → ohne Aufrufparameter
- Copy → mit einer const Referenz auf eine andere Instanz
- sonstige → werden anhand ihrer Aufrufparameter unterschieden, Sprechweise:`überladen`

Werden verschiedene Konstruktoren definiert, muss der Default zuerst aufgerufen werden, wenn dieser erhalten bleiben soll.

3.1.2 Implizit

Falls ein expliziter nicht angegeben wurde, dieser jedoch aus technischen Gründen benötigt wird:

- Default → macht nichts, alle Parameter bleiben uninitialisiert
- Copy → dieser kopiert alle Attribute der anderen Instanz eins-zu-eins(bytewise). Problematisch, wenn die Instanz Pointer auf etwas hat.
- ...

Wenn ein Objekt einen Pointer auf einen allokierten Speicher hat, muss eine spezielle Copy Methode verwendet werden. Der Pointerwert / Adresse wo dieser hinzeigt, wird einfach kopiert. Falls dann das erste Objekt den Speicher freigibt, wird die Speicheralokation gelöscht. Sobald Objekt 2 nun den Speicher aufruft, zeigt dieser auf ungültigen Speicher → *Segmentation fault*.
Darum **MUSS** dem Objekt ein separater Speicherbereich gegeben werden.

Merke: Konstruktoren werden immer dann gerufen (evtl. auch **implizit**) wenn ein neues Objekt in den Speicher gelegt wird.

3.2 Destruktor

Ein Destruktor eintfernt eine Instanz aus dem Speicher, hat keine Aufrufparameter und auch kein Rückgabewert. Es kann immer nur **einen** Destruktor geben. Wenn der Destruktor fertig ist, sollte kein Speicher mehr vom Objekt belegt sein.
Destruktoren sollten immer **virtuell definiert werden** (siehe 10.2.1). Mit anderen virtuellen Funktionen **müssen** sie virtuel definiert werden. Der Funktionsname beginnt mit einer Tilde(~). Der Destruktor wird evtl. auch Implizit aufgerufen z.b., wenn aus einer Funktion wieder herausgesprungen wird.

3.3 Beispiel

```
#include <iostream>
class Storage{
    public:
        Storage();
        virtual ~Storage();
        void add(int in);
        void nix(Storage inC);
    private:
        int* data;
        int size;
};//eine Klasse die Int Werte speichert, aehnlich wie ein array

// ---- End h File

Storage::Storage(){ // Konstruktor
    data = nullptr;
    size = 0;
}



Storage::~Storage(){ // Destruktor
    delete[] data;
    data = nullptr;
    size = 0; // Speicher der allokiert wurde sollte hier freigegeben werden
}
void Storage::add(int in){
    // todo
}
void Storage::nix(Storage inC){
    // todo
}

// ---- End cpp File

int main(){

    Storage* i1 = new Storage; // default konstruktor
    Storage i2; // default konstruktor
    i1->nix(i2); // Call-by-Value. Eine Kopie von i2 wird auf den Stack gelegt -> Copy-Konstruktor!

    delete i1; // destruktor von i1 wird explizit aufgerufen
    return; // destruktor von i2 wird implizit aufgerufen
}
```

4 Initialisierungslisten und direkte Initialisierung

Die Initialisierung von Datentypen kann auf verschiedene Arten erreicht werden. Es wird zwischen Zuweisung und Initialisierung unterschieden.

4.1 POD's

`plain old datastructure` / built in datatypes

```
// Zuweisung fuer POD:
int i; // Speicher fuer Instanz wird alloziert, Wert ist uninitialisiert
i = 5; // Wert (aus anderer Speicherstelle) wird zugewiesen

// Initialisierung fuer POD: Kein weitere Speicherverbrauch
int i = 5;  // (C-Style)
int i(5); // (C++-Style bis C++11)
int i{5}; // (neuer C++-Style)
// Aus Effizienzgruenden bevorzugen wir (fast) immer die (direkte) Initialisierung!
```

4.2 Non PODs

```cpp
Aclass m;//kein Initialwert
// Vorsicht: Falls "Aclass" gross ist, muss viel kopiert werden
m = otherInstance;

// Copy-Initialisierung fuer non-POD:
Aclass m = otherInstance; //(copy ctor)
Aclass m(otherInstance); // C++-Schreibweise
Aclass m{otherInstance}; // C++-Schreibweise seit c++11

//Besser: direkte Initialisierung auch fuer non-PODs:
Aclass m("Eagle 1", 1, 2, ...); // (vor C++11)
Aclass m{"Eagle 1", 1, 2, ...}; // bevorzugte Schreibw. seit C++11
// Die Instanz wird ohne Umwege mit den gewuenschten Werten in den Speicher geschrieben, sofern ein geeigneter user-defined ctor vorhanden ist!
```

4.3 User defined Constructor

Um eine Klasse richtig initialisieren zu können muss der Ctor korrekt definiert werden. Eine sogenannte Initialisierungsliste wird verwendet:

```cpp
student::student(int _semester,
                 long int _eltVerzweiflung) :
    semester{_semester},
    eltVerzweiflung{_eltVerzweiflung}
{//die Reihenfolge der Initialisierung ist durch die Reihenfolge im Speicher gegeben!
    //Rumpf kann leer bleiben Initialisierung bereits fertig
}
```

5 Assertions

Assertions sollten Annahmen über korrekte Wertebelegung darstellen. Sie sind kein C spezifisches Konzept und sollten **nur zu Testzwecken** eingesetzt werden. Im Releasebuild sollten sie deaktiviert werden. Dafür gibt es spezifische Präprozessorflags. Der Header <**assert.h**> bzw. <**cassert**> stellt hierfür ein Präprozessor-Makro `assert(Bedingung)` zur Verfügung, um solche Zusicherungen auszudrücken.
Beispiel:

```cpp
#include <cassert>
//#define NDEBUG // Deaktiviert Assertions im Projekt
bool testStuff(int* in1){

    assert(in1 != nullptr);
    //assertion

    ;//restliche implementation einer Funktion

}
```

Das obige Beispiel hat eine Assertion. Diese würde im Test, falls ein Nullpointer übergeben wird ein Fehler ausgegeben.

6 This-Pointer

Mit dem `this Pointer` kann ein Pointer des aufrufenden Objekt zurückgegeben werden. Bsp:

```cpp
class Stuff{
    public:
        Stuff& funktion(int valIn);
    private:
        int value;
};
Stuff& Stuff::funktion(int valIn){
    //do Stuff with val z.b. addieren auf interne Variabel
    this->funktion(0);// auch moeglich
    return *this;//this pointer -> wird dereferenziert und Referenz zurueckgegeben
}
int main(){
    Stuff memb;
    memb.funktion(1).funktion(2).funktion(3);
}
```

Nacheinander werden diese Funktionen durchlaufen da Funktion(2) vom zurückgegebenen Pointer aus Funktionsaufruf 1 aus gerufen wird. Die funktion wird **Kaskadiert**

7 Overloading

Operator Overloading bedeutet das bestehende Operatoren **überladen** werden im Sinne neu definiert / darüber geladen. Sie erhalten **neue Bedeutung** für eine Klasse.

7.1 Methodenoverloading

Methoden dürfen in Cpp gleich heissen, sie müssen aber unterschiedliche Aufrufparameter haben.

```cpp
int addiere(int v1, int v2);
double addiere(double d1, double d2);

int i = addiere(2, 7); // ruft addiere(int, int)
double d = addiere(2.3, 2.0);// ruft addiere(double, double)
```

7.2 Operator overloading

Es können die inkludierten Operatoren von Cpp überladen werden. Erlaubte Operatoren sind:

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

Nicht erlaubte Operatoren sind: `. .* :: und ?:`.
Es wird zwischen overloading global und in Methode unterschieden.

7.2.1 Global

Globales overloading funktioniert mit einer als `friend` deklarierten Funktion. Als friend deklarierte Funktionen agieren als **globale Funktion** und haben vollen zugriff auf **alle Parameter einer Klasse**. Sie sind daher zu vermeiden da sie eher ein Workaround sind und zu Problemen führen können.

Syntax : `<returntype> operator<type>(type val1, type val2);`
BSP:

```cpp
// H file
class Dozent{
    friend Dozent& operator+(Dozent& in1, Dozent& in2);
                //der + Operator wird ueberschriben
    private:
        int numb;
};

Dozent& operator+(Dozent& in1, Dozent& in2);

// ------ start cpp ------
Dozent& operator+(Dozent& in1, Dozent& in2){
        in1.numb += in2.numb;// whatever Operation welche zum + Operator passt
        return in1;
}
```

Die Funktion kann von überall aus dem Programm ausgeführt werden. Es sollte nur dann verwendet werden, wenn der erste Parameter (links) nicht eine Instanz der Klasse ist da der erste Aufrufparameter immer die Instanz sein muss.

7.2.2 Methoden

Ein Operator kann auch als Methode einer Klasse definiert werden. Sie haben nur noch ein Aufrufparameter.
Syntax : `<returntype> operator<typ>(type in);`
Der 2. Parameter ist das Objekt selbst, welche die Methode aufruft.

```cpp
// H file
class Dozent{
    friend Dozent& operator+(int in1, Dozent& in2);
    public: // friends immer vors public
        Dozent& operator+(Dozent& in1);//der + Operator wird ueberschriben(Falls eine andere Instanz addiert wird)
        Dozent& operator+(int in1);//der + Operator wird ueberschriben(falls ein int addiert wird)
    private:
        int number;
};
Dozent& operator+(int in1, Dozent& in2);//dieser Fall muss global / mit friend geloest werden

// ------ start cpp ------
Dozent& operator+(int in1, Dozent& in2){
        in2.number += in1;
        return in2;
}
Dozent& Dozent::operator+(Dozent& in2){
    number += in2.number;
    return *this;
}
Dozent& Dozent::operator+(int in2){
    number += in2;
    return *this;
}

// ------ start main file ------
main(){
    Dozent D1;
    Dozent D2;
    D1 + D2;// fall mit anderer Instanz
    D1 + 1;// fall mit int
    1  + D1;// durch Global overloading abgedeckt
}
```

Die obigen Funktionen machen alle dasselbe. Sie unterscheiden sich durch die **unterschiedlichen Argumente**. Diese decken verschiedene Eingabetypen ab. Es wäre schlechter Stil, die Typkonvertierung dem Compiler implizit zu überlassen! Ein Fall,

bei dem das anderstypige Argument zuerst kommt, muss noch per Globalem Overloading gelöst werden. Da der erste Aufrufparameter, beim Overloading mit Methoden, immer zuerst kommt.

8 Default Argumente

Default Argumente werden als Parameterwert verwendet, wenn keiner mitgegeben wird. Parameter werden von links nach rechts mit Werten belegt. Wenn manche Aufrufparameter keinen Default wert erhalten, müssen diese weiter Links stehen, als welche mit Default wert.

```
void func(int a = 0, int b = 0, int c = 0){}
func();         // a = 0, b = 0, c = 0
func(1);        // a = 1, b = 0, c = 0
func(1, 1);     // a = 1, b = 1, c = 0
func(1, 1, 1);  // a = 1, b = 1, c = 1
//Alle Funktionsaufrufe valide

void func2(int a, int b = 0, int c = 0); // korrekte
     Reihenfolge
```

Default Argumente können überall verwendet werden bis auf Aufrufparameter von Operatoroverloading. Dort sind sie **nicht** erlaubt. Dort machen sie aber ohnehin keinen Sinn.

9 Getter & Settermethoden

Getter und Settermethoden werden Typischerweise verwendet um Lese und Schreibzugriffe auf eine Klasse zu regeln. Attribute sind **alle Parameter einer Klasse**. Mit einer Get oder Set Methode kann der Zugriff auf diese kontrolliert / angepasst werden.

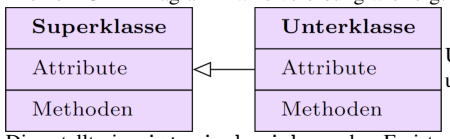
```
#include <string>
class Stuff{
    public:
       const std::string& getName() const;
    protected://erlaubt schreibenden Zugriff von Unterklassen
       void setName(const std::string& in);
    private:
       std::string name;//privates Attribut
};
const std::string& Stuff::getName() const{ // Getter
    return name;
}
void Stuff::setName(const std::string& in){ // Setter
    name = in;
}
```

10 Vererbung

Vererbung erlaubt es Attribute und Methoden von anderen Klassen zu übernehmen. Die Oberklasse, Baisklasse oder Superklasse `vererbt` an eine Unterklasse, derived class oder eine Spezialisierung. Bei der Vererbung werden immer **alle** Attribute oder Methoden weitergegeben.

10.0.1 UML

In einem UML Diagramm wird vererbung wie folgt dargestellt:



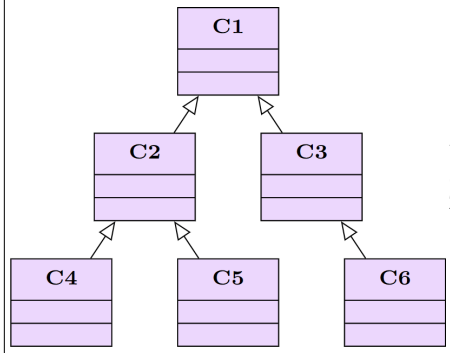
Unterklasse erbt (die Attribute und Methoden) von Superklasse.

Dies stellt eine `ist ein beziehung` dar. Es ist sehr wichtig, dass die Pfeilspitze **genau so!** aussieht wie in der Darstellung da ansonsten Verwechslungsgefahr mit anderen Mechanismen entstehen könnte.

10.0.2 Syntax

```
class Unterklasse : public  Superklasse{
    ...
} ; // Unterklasse erbt Superklasse
```

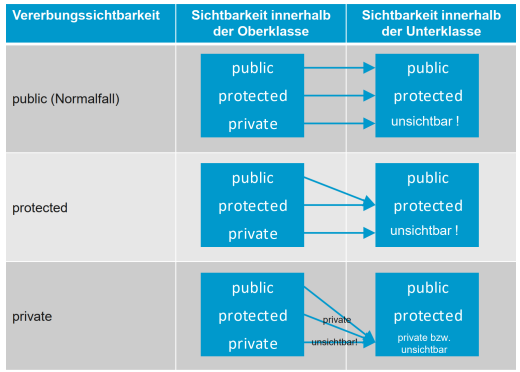
10.0.3 UML++



Vererbung ist auch über mehrere Stufen Möglich:
Z.b. C4 Erbt alles von C2 und C1

10.0.4 Sichtbarkeit

Die Sichtbarkeit ist durch das Public definiert. Es ist möglich die Sichtbarkeit aller Attribute und Methoden einzuschränken. Mit **Public** wird alles mit der originalen Sichtbarkeit übernommen. Mit **protected** wird alles was public war protected. Mit **private** wird alles private (nicht sehr nützlich).



Wird eine Vererbung private durchgeführt, sind keine Attribute oder Methoden mehr sichtbar.

10.0.5 Constructor / Destructor chaining

Grundsätzlich initialisieren ctors nur die eigene Klasse. Dh Superklassen initialisieren sich selbst. Eine Subklasse initialisiert sich, indem sie zunächst einen Ctor der Superklasse aufruft und danach sich selbst **IMPLIZIT**. Man kann das auch explizit machen:

```
Unterklasse::unterklasse(int _initvalOberklasse,
                         int _initvallUnterklasse):
    Oberklasse{_initvalOberklasse},// muss zuerst stehen da
     diese zuerst initialisiert werden muss
    unterklassenAttribut{_initvallUnterklasse}
{} // bleibt leer
```

Note: Da der Konstruktor aufgerufen wird kann auch ein evtl Private Attribut der Oberklasse initialisiert werden. Gleiches gilt mit dem Dtor, nur umgekehrte Reihenfolge (erst wird der Dtor von der subklasse aufgerufen, dann.....).

10.1 Überschreiben von Methoden

Geerbte Funktionen können `überschrieben` werden. Die Unterklasse definiert eine neue Methode mit demselben Namen. Allerdings kann es dann zu Mehrdeutigkeit kommen:
Im folgenden Beispiel haben eine Subklasse und eine Oberklasse eine print Funktion welche **nicht** virtual gekennzeichnet ist:

```
int main() {
    Subclass p;             // Klassenerstellung
    p.print();              // print von Subclass, ok
    Subclass* sPtr = &p;    // Subclass Pointer
    sPtr->print();          // print von Subclass, ok
    Topclass* tPtr = &p;    // !! Topclass Pointer !!
    tPtr->print();          // print von Topclass!!!!!!
    Subclass& sRef = p;     // Subclass ref
    sRef.print();           // print von Subclass, ok
    Topclass& tRef = p;     // !! Topclass ref !!
    tRef.print();           // print von Topclass!!!!!!
}
```

Dieses Beispiel soll zeigen, dass Pointer, welche eigentlich von einer Oberklasse auf eine Unterklasse zeigen auf die `Eigenen` Funktionen der Oberklasse zeigt, solange diese nicht überschrieben worden sind. Dies kann als Verhalten gewünscht sein.

10.2 virtual, final, override und scopeoperator

10.2.1 virtual

Mit **virtual** kann eine Methode gekennzeichnet werden, dass diese Methode von einer Unterklasse überschrieben wird. Somit würde das obige Beispiel nicht mehr funktionieren und es wird immer die Printfunktion der Unterklasse aufgerufen. In einer Unterklasse ist virtual nicht mehr zwingend zu schreiben. Eine Klasse mit einer virtuellen Funktion wird als Polymorph bezeichnet.

10.2.2 Virtual beim Dtor

Ist eine Methode als virtual deklariert, so muss der Dtor **ZWINGEND** auch als virtual deklariert werden.

10.2.3 Override

Soll eine Methode eine geerbte Methode ersetzen so muss diese mit `Override` gekennzeichnet werden. Solch eine Methode ist automatisch auch virtual. Der Compiler kann so überprüfen, ob eine virtual Methode vorhanden ist.

10.2.4 Final

Final kann zum einen verbieten, dass eine Methode einer Klasse überschrieben wird (Logischerweise geht das nur bei virtuellen Methoden). Zum anderen kann sie auch das weitere Erben einer Klasse verbieten.

10.2.5 Scopeoperator

Per Scopeoperator kann (::) eine Methode aus einer Oberklasse gezielt aufgerufen werden. Mit diesem wird garantiert die Funktion welche definiert wird aufgerufen, egal ob diese aufgrund virtual überschrieben wird.

```
class Subexample : public Uppclass{
    virtual ~Subexample(); // Correct Dtor
    virtual void newMethod(); //gets overriden when inherited
    // void oldVirtualMethod(); this inherited func doesnt get
      changed, it stays virtual
```

```
    void getsOverridden() override;// overrides method in
     uppclass
    virtual void iAmPerfect() final;// cant be overridden
};
void Subexample::newMethod(){
    Uppclass::aMethodfromUpclass();
    //calls a class from a class higher in the chain
}
```

10.2.6 RTTI / Typing / Binding

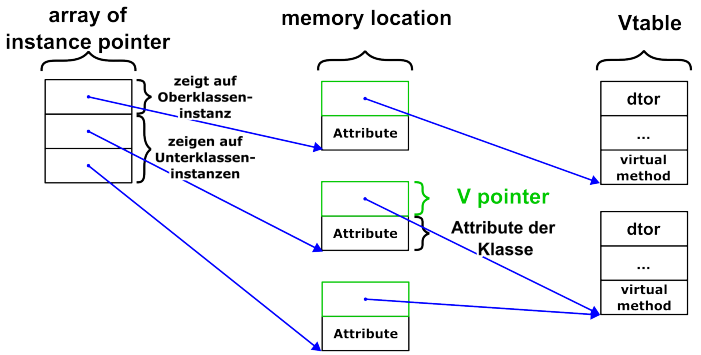
Wird kein Virtual verwendet, nennt man das **static binding**. Mit virtual wird es **dynamic binding** genannt, da erst bei Laufzeit bekannt ist, um welchen Pointer es sich handelt.
Um bei Laufzeit herauszufinden, um welchen Typ es sich handelt kann die **Run-time Type Information** verwendet werden. Dies geschieht mit der **typeid()** Funktion aus dem Header <**typeinfo**>
typeid gibt ein std::type_info zurück.

```
#include <typeinfo>
int main() {
    int a{0};
    float b{0.0f};
    if (typeid(a) != typeid(b)) {// is always true
    }// because they have a different type
}
```

10.2.7 Implementation von RTTI

Das RTTI System arbeitet meistens mit einem so genannten V-Table in dem zu allen(evtl geerbten) virtuellen Methoden ein Pointer definiert ist. Die Klasse erhält zusätzlich einen V-table Pointer. Zur Laufzeit kann die Instanz via V-table die korrekte Methode ermitteln.

Das folgende Beispiel zeigt das die Memory-map von einem Array mit 2 Oberklassenpointer. Ein Pointer zeigt auf eine Oberklasse, die anderen auf Unterklassen, welche virtuelle Methoden überschrieben haben. Der V-pointer der Ober- und Unterklasse ist darum unterschiedlich / sie zeigen nicht auf denselben Vtable. Die Unterklassen zeigen wiederum auf denselben Vtable.



10.3 Abstrakte Klassen

Wenn Klassen zwar festlegt, dass eine Methode da ist, diese aber noch nicht implementieren, nennt man das eine abstrakte Methode. Man spricht dann von einer abstrakten Klasse. Eine abstrakte Methode wird in UML ***kursiv*** dargestellt. Es ist egal, ob die abstrakten Methoden selbst definiert sind oder geerbt. Abstrakte Klassen kann man nicht instanziieren. Es gibt kein spezielles Schlüsselwort dafür, man weist der Methode bei Definition 0 zu:

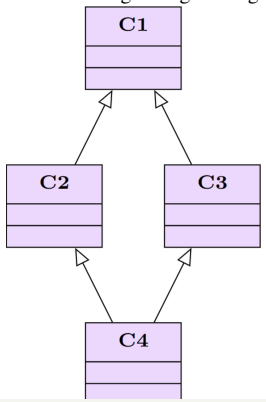
```
class ModernArt{// abstrakte klasse (wegen abstrakter Methode)
    virtual void aMethod() = 0;// abstrakte Methode
};
```

10.4 Organisation der H files

Generell gilt immer noch: Jedes H File hat ein Cpp File. Allerdings können Unterklassen in ein H File zusammengefasst werden, falls diese nicht allzu Umfänglich sind. Ansonsten sollte ein neues H File erstellt werden. Für abstrakte Klassen fällt die Implementierung in einem Cpp file weg.

10.5 Mehrfachvererbung

Folgender Vererbungszweig ist möglich:



Solche Gebilde sind zwar möglich, sollten jedoch vermieden werden, da schnell sehr kompliziert und kaum lesbar. Es entsteht schnell Mehrdeutigkeit durch die verschiedenen Erbzweige.
Bei der Definition einer Klasse können weitere Oberklassen mit einem Komma getrennt angegeben werden :

```
class Unterklasse : public Oberklasse1, public Oberklasse2{};
```

10.6 Static

Static im Zusammenhang mit Klassen bindet eine Methode oder ein Attribut an die Objektklasse und nicht an die Objektinstanz (ähnlich zu Python Klassenvariablen).

10.6.1 Methoden

Static Methoden müssen im Header-File deklariert und im Cpp-File definiert werden. Sie dürfen nur auf Static definierte Attribute zugreifen da keine Objektinstanz vorhanden. Hauptanwendungen sind Hilfsfunktionen wie Funktionsaufrufcounter oder Umrechnungsfunktionen. Bei der Deklaration wird der Klassennamen verwendet (siehe Bsp).

10.6.2 Attribute

Static Attribute müssen im H deklariert und im Cpp File definiert werden. Sie werden verwendet für z.B. Objektspezifische Konstanten oder im Zusammenhang mit Static Counter.

10.6.3 Bsp

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

// End H-File----

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

11 Memorymanagement

Es gibt verschiedene Gründe warum nicht immer gleich viel Speicher verwendet wird z.B. da nicht unbegrenzt vorhanden ist. Speicher wird nur dann verwendet, wenn er wirklich gebraucht wird. Das Memorymanagement muss aktiv beachtet werden. Es wird zwischen automatischen und manuellen Speichermanagement unterschieden.

11.1 Automatisch

Automatisches Speichermanagement wird anhand der Lebenszeit einer Variable festgestellt und während dem Programmieren bereits festgelegt. Der Speicher liegt auf dem sogenannten **Stapel / Stack**.
Wird eine Funktion aufgerufen, werden die benötigten Variablen auf dem Stack definiert. Ebenso wird eine Returnadresse zur Funktion, die die Funktion aufgerufen hat abgelegt. Wie die Daten angeordnet werden, ist Architektur-abhängig. Wird die Funktion wieder verlassen, werden die Variablen vom Stack wieder entfernt und zur aufrufenden Funktion zurückgesprungen.

11.2 Manuell/dynamisch

Man kann auch manuell Speicher anfordern. Zum Beispiel, falls zur Compilezeit nicht bekannt ist, wie viel Speicher gebraucht wird. Dieser Speicher ist auf dem sogenannten **Haufen / Heap**.
Der Heap ist ein Speicherbereich, der vom Betriebssystem bereitgestellt und verwaltet wird.
Zur Laufzeit wird Speicher dynamisch angefordert. Dieser bleibt so lange belegt bis dieser Manuell wieder freigegeben wird.
Es kann aber auch sein, falls der Speicher stark fragmentiert ist, das kein Speicher bereitgestellt werden kann. Generell ist bei sehr vollem Speicher das Arbeiten erschwert.

11.2.1 Speicherlecks

Falls ein Programm unkontrolliert Speicher aufnimmt oder den ihm zur Verfügung gestellte nicht wieder freigibt, spricht man von einem Speicherleck. Diese sind zu verhindern da diese zu einer Systemüberlastung und Abstürzen führen.

11.3 Speichermanagement funktionen

Es gibt einige Funktionen für Speichermanagement. In C und C++ sind diese leicht verschieden. Generell geben diese einen Pointer zurück, welcher auf freien Speicher zeigt. **Diese müssen immer auf einen Nullpointer geprüft werden, da keine Garantie vorhanden ist, ob überhaupt Speicher vorhanden ist!** Generell sollte mit solchen Pointer immer misstrauisch gearbeitet werden und nach Abschluss immer wieder freigegeben werden.

11.3.1 C: malloc( ) calloc( ), free( ) (Funktionen)

malloc() gibt einen Voidpointer(pointercasting nicht vergessen) zurück welche auf eine angeforderte Grösse an Bytes Speicher zeigt. Der Speicher ist **nicht** initialisiert
calloc() macht dasselbe, initialisiert aber den Speicher auf 0.
free() gibt den Speicherbereich wieder frei.

```
#include <stdio.h>
#include <stdlib.h>

void *malloc(size_t size);
// Alokiert Speicherbereich size in Bytes
void *calloc(size_t size);
// Alokiert Speicherbereich size in Bytes und setzt diesen 0
free(void *ptr);
// Gibt Speicherbereich auf welcher ptr zeigt wieder frei

```

```c
int main(){
    int* iPtr = NULL;
    iPtr = (int*)malloc(3*sizeof(int));
    // Speicher fuer 3 Ints reservieren
    // Insert Nullpointer check here!!!
    free(iPtr);
}   // Speicher wider freigegen
```

Falls kein Speicher allokiert werden kann, wird ein NULL-pointer zurückgegeben.

11.3.2 C++: new, delete, new[ ], delete[ ] (Operatoren)

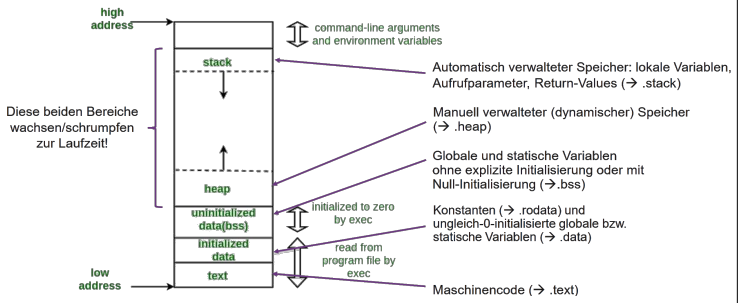
Der **new**-Operator erstellt ein Pointer welcher auf die mitgegebene Grösse zeigt. **new[ ]** macht dasselbe, ausser das dieser ein Array zurückgibt.
delete / delete[ ] gibt den Speicher wieder frei. Dieser kann auch auf den Nullpointer ausgeführt werden. Dieser ist `Delete` egal.

```c

int main(){
    int* intPtr = new int;
    // Allokiert fuer intPtr ein Speicherbereich eines Ints
    int* intArrayPtr = new int[10];
    // Allokiert fuer intArrayPtr ein Speicherbereich eines
     Int Array der groesse 10
    if(intPtr == nullptr) return-1;// Nullpointercheck
    delete intPtr;
    // Gibt intPtr wieder frei
    delete[] intArrayPtr;
    // Gibt intArrayPtr wieder frei
}
```

11.4 Speicheraufbau

Der Speicheraufbau bildlich dargestellt sieht ungefähr so aus:



Je nach Architektur kann es auch sein, dass die hohen und tiefen Adressen anders herum sind.

11.5 Referenzen

Referenzen sind die modernere / eingeschränkte Version von Pointer. Wenn möglich sollte immer mit Referenzen gearbeitet werden, dies ist aber nicht immer möglich.

Referenzen...
- wirken wie ein Alias-Name einer Variablen
- werden wie normale Variablen verwendet
- sind niemals uninitialisiert
- haben niemals den Wert nullptr
- brauchen nicht immer Speicher (Implementations abhängig)

Generell helfen Referenzen weniger Fehler in der Programmierung zu machen und weniger Risiken zu haben. Pointer werden allerdings weiter für sehr hardwarenahe Programmierung gebraucht. **sizeof()** liefert die Grösse des Typs, auf der die Referenz zeigt.
Beispiel:

```c
int i = 42;

// myref ist eine Referenz auf i:
int& myref = i;

// kann auch bei einem Funktionsaufruf verwendet werden:
void myfunc(const DaClass& in1,int& in2);
// hier werden in1& 2 als referenz uebergeben. "DaClass" ist
     hier auch Schreibgeschuetzt (wie ein constpointer)
```

11.6 Speicherplatzbedarf von Objekten

Eine Unterklasse braucht immer mindestens so viel speicher wie eine Oberklasse.

11.6.1 Zuweisungen

Zuweisungen von Oberklasse zu Unterklasse sind in der regel möglich da alle geerbten Attribute ja vorhanden sind. Umgekehrt geht das allerdings nicht, da der Speicher dafür nicht initialisiert ist.

11.7 Type Casting

Cast:	Anwendungsbereiche	Wissenswertes	Beispiele

implizit: kein Cast	• Numerische Typen u. bool (ggf. mit Warnung, falls Wertebereich möglicherweise nicht ausreicht)
	• Objektinstanzen einer Unterklasse in einer Variablen vom Typ der Oberklasse speichern (Upcast).
	• Pointertypen, deren Umkonvertierung «ungefährlich» ist – die genauen Regeln sind ziemlich kompliziert.
			1. signed char a = 1;
			int b = a;
			
			2. int a = 1;
			char b = a; // ggf. mit Warnung
			
			3. Bird b;
			Animal a = b; // Upcast: a ist ein Animal

static	• Alles, was implizit möglich ist, unterdrückt dabei etwaige Warnungen.
	• Pointer- und Referenz-Typen auf Instanzen von Ober- und Unterklassen in beide Richtungen umwandeln: Upcast und Downcast (**ohne Typprüfung** zur Laufzeit, Auswertung ausschliesslich zur **Compilezeit**)
			1. int a = 1;
			char b = static_cast<char>(a);
			2. Bird* b = new Bird;
			Animal* a = b; // geht implizit!
			Bird* c = static_cast<Bird*>(a);
			/* Braucht einen down-Cast , weil nicht jedes Animal ein Bird ist!*/

dynamic	• Pointer- und ReferenzTypen auf Instanzen von **polymorphen** Ober- und Unterklassen in beide Richtungen umwandeln: Upcast und Downcast (**mit** Typprüfung per RTTI zur Laufzeit)	Unterstützt ausserdem einen Upcast für Pointer- und Referenztypen auf nichtpolymorphe Klassen (normalerweise machen wir dies aber implizit oder verwenden einen static_cast!) Falls der dynamic_cast **fehlschlägt**, liefert er bei Pointer-Typen den Wert **nullptr**, bei Referenzen wird eine **std::bad_cast-Exception** geworfen	Animal* a = new Ant; Bird* b = dynamic_cast<Bird*>(a); /* Der Dynamic Cast wird immer zur Laufzeit ausgewertet, dh: b ==nullptr */

const	verändert die `constness` und/oder `volatile-ness` des Ziels eines Pointer oder Referenz Typs.		const int val = 10;
			const int *ptr = &val;
			int *ptr1 = const_cast <int *>(ptr);

reinterpret	• Konvertiert zwischen beliebigen Pointer-Typen sowie zwischen beliebigen Pointer und int-Typen **ohne Typprüfung**. • Kann auch mit Referenz-Typen umgehen.	Achtung: erzeugt plattformspezifisches Verhalten	Animal* a = new Ant; uint8_t* b = reinterpret_cast<uint8_t*>(a);

Die C Syntax für Casting sollte nicht mehr verwendet werden da meistens nicht optimales verhalten.

11.8 Templates

Templates erlauben es Typen-unabhängigen Code zu schreiben. Eine Funktion wird mit einem **Platzhalter-typ** geschrieben. Bei der Verwendung wird ein oder mehrere zu verwendende Typen mitgegeben (es ist auch möglich Konstanten mitzugeben). Es sind Klassen und Funktionstemplates möglich. Templates werden immer zur **Compilezeit** ausgewertet. Templates brauchen eine Deklaration, Definition und Instanziierungsparameter. Sie werden ausschliesslich in einem H-File deklariert.
Achtung! Templates werden für jeden **unterschiedlichen** Typen der Instanziiert wird angelegt. Das kann zu hohem Speicherbedarf führen.

11.8.1 Implementation

Eine Funktion / Klasse kann als Template definiert werden indem `template<typename1 T1, typename2 T2, ...>` (mit<>) vor die Funktion / Klasse geschrieben wird. Die Typendefinitionen sind dann durch das T zu ersetzten.

```
// H-File
template<typename T>
void swap(T& a, T& b); // deklaration

template<typename T> // definition
void swap(T& a, T& b) {
    T temp{b};
    b = a;
    a = temp;
}
```

11.8.2 Syntax

```
// Template-Definition fuer Funktionsdeklaration:
template<typename T1, typename T2, ...>
returntype name(type1 param1, type2 param2 , ...);

// Template-Definition fuer Funktionsdefinition:
template<typename T1, typename T2, ...>
returntype name(type1 param1, type2 param2 , ...) { ... }

// Klassentemplate:
// Template-Definition fuer Klassendeklaration:
template<typename T1, typename T2, ...>
class Classname{ ... };

// Template-Definition fuer Methode einer Klasse:
template<typename T1, typename T2, ...>
returntype Class-name<T1, T2, ...>::method-name(type1 param1,
     type2 param2 , ... ) { ... };
```

11.8.3 Instanziierung

Klassentemplates (<> sind nicht fakultativ):
`Templatenamename<type1, type2, ...>Instanzname`

Funktionstemplates (<> sind nicht fakultativ):
`Templatenamename<type1, type2, ...>(param1, ...);`

Falls bei der Instanziierung des Funktionstemplates der Typ bereits klar ist, können diese weggelassen werden.

12 Exceptions

Exceptions werden zur Behandlung von Fehlern verwendet. Eine Exception ist eine Instanz eines beliebigen Datentyps, welche Fehlerinformation transportiert. Eine Exception wird bei Fehlerauftritt `geworfen` (throw), und zur Behandlung `gefangen` (catch).

```
try {
    // dangerous Code with throw
} catch (someType someName) {
    // some error handling.
}
// normal execution continues here
```

Diese Syntax ist so in der Praxis wenig aufzufinden da das Fehler werfen meist in einer Unterfunktion gemacht wird. Eine geworfene Exception wird dann mit code um die Instanziierung gehandhabt. Dies erlaubt es Fehler von Unterfunktionen je nach Situation anders zu handhaben:

```
#include <iostream>

int divide(int dividend, int divisor) {
    if (divisor == 0) throw "Division by zero!";
    return dividend/divisor;
}
int foo(int a, int b) {
    std::cout << "Entering foo()" << std::endl;
    int result = divide(a, b); // can throw exception
    std::cout << "Leaving foo()" << std::endl;
    return result;
}

int main() {
    try {
        std::cout << foo(7,0) << std::endl;
    } catch (const char* e) {
        std::clog << "Not optimal but ok, " << e << std::endl;
    }// just writes the Error to the log

    // continue Program Function
    std::clog << "Continue Programm" << std::endl;

    try {
        std::cout << foo(12,0) << std::endl;
    } catch (const char* e) {
        std::cerr << "Critical error: " << e << std::endl;
        std::cerr << "Program terminating" << std::endl;
        exit(-1);
    }
    std::clog << "finish Programm" << std::endl;
    // continue Program Function
}
```

Cout:

- Entering foo()
- Not optimal but ok, Division by zero!
- Continue Programm
- Entering foo()
- Critical error: Division by zero!
- Program terminating

12.1 Mehrere Catch Blöcke

Unterschiedliche Exceptions können mit mehreren seriellen Catch Blöcken erreicht werden. Wenn der letzte Catch block mit `...` aufgerufen wird, werden alle Typen Exceptions gehandelt.

```
int main() {
    int x;
    std::cin >> x;
    try {
        a(x); // can throw various types of exceptions
    } catch (const char* e) {// handles only Type const char*
        std::clog << e << std::endl;
    } catch (int i) {// handles only Type int
        std::cerr << i << std::endl;
    } catch (...) {// handles whatever...
        std::cerr << "other exception occured" << std::endl;
    }
return 0;
```

12.2 Fehlender catch block

Falls kein Passender catch Block gefunden werden kann wird die Funktion std::terminate() aufgerufen. Diese ruft den registrierten **std::terminate_handler** (Funktionspointer). Defaultmässig ist **std::abort()** hinterlegt welche das Programm beendet ohne Destruktoren zu rufen (crash). Falls dieses Verhalten angepasst werden soll, kann man das mit `set_terminate(std::terminate_handler f)`.

12.3 leerer Catch block

straight to hell

12.4 noexcept

Um zu signalisieren, ob eine Funktion eine Exception auslösen kann oder nicht, kann der specifier **noexept** verwendet werden. Dafür ist aber **der Entwickler zuständig**. Der Compiler kann **nicht** prüfen, ob die Funktion tatsächlich eine Exception wirft oder nicht.

12.5 <stdexcept>

stdexcept ist dazu da um einen einfachen Austausch von Fehlerinformationen vorzunehmen. Dafür sind diverse standardisierte Fehlertypen als Klassen definiert. Z.b.:
std::logic_error **std::out_of_range** **std::overflow_error** Für Details siehe :
https://en.cppreference.com/w/cpp/header/stdexcept

13 std:vector

std:vector ist ein Klassentemplate welches die Kapazität selbständig bei Bedarf erweitert.

```
#include <vector>
std::vector<dt> v;// Template instanzieren mit <datentyp> (dt)
v.push_back("Value");// Value anfuegen
v.erase("iterator");// Elemente loeschen(Iterator=fancy
    Pointer)
v.begin();// returned ein Pointer auf das erste element
v.end();// returnd ein Pointer HINTER das letzte Element

for (const dt s : v) { // Iteriert ueber Elemente vom Vector
    doSomething(s);} // do something mit element

for (std::vector<dt>::iterator it=v.begin();it!=v.end();it++)
    {
    doSomething(s);} // do something mit element, altes for
```

14 Makefiles

Makefiles sollten generell das Umsetzten des Codes in Maschinencode vereinfachen. Es ist eine `Skriptingsprache` welche grob nach dem Muster `Erzeugnis/target : Abhängigkeiten/Dependency` folgt. Dann folgt indentiert der Befehl, um dieses Erzeugnis zu erzeugen.
Wenn ein Erzeugnis keine Datei zurückgibt, muss dieser als `.PHONY` markiert werden. Das verhindert, dass eine Datei mit demselben Namen das Ausführen verhindert.
Am besten sieht man das an einem Beispiel:

```
all : stuff                 # erzegt die Definition fuer make
    all
```

```
2
3  project : main.o lib.o      # Projekt linken
4    clang++ -Wall -o vector Vector3D.o main.o
5
6  lib.o : lib.cpp             # lib.cpp compilen
7    clang++ -Wall -c lib.cpp
8
9  main.o : main.cpp           # main.cpp compilen
10   clang++ -Wall -c main.cpp
11
12 clean :                     # Projekt cleanup
13   rm -f project.exe lib.o main.o
14
15 .PHONY : clean all          # Mariert die Targets "clean" und
       "all" als "PHONY" -> Lesbarkeit
```

Der Befehl `make all` baut nun das Projekt.
Sind die o Dateien noch aktuell werden diese nicht erneut kompiliert. Das Projektverzeichnis kann einfach durch `make clean` aufgeräumt werden.

14.1 Platzhalter

In einem Makefile können auch Platzhalter verwendet werden:

- $ @ Dateiname des Targets
- $ < Dateiname der ersten Dependency des aktuellen Targets
- $ ∧ Dateiname aller Dependencies des aktuellen Targets durch Leerzeichen getrennt

Mit diesen Mitteln kann ein Professionelleres Makefile erstellt werden, welches relativ universell eingesetzt werden kann:

```
1  CXX = clang++               # Verwendeter Compiler
2  CXXFLAGS = -Wall -c         # compilerflags
3  LDFLAGS = -o                # loader flags
4  BIN = main                  # binary output / Name des .exe
       file
5  OBJS = drink.o drinktest.o  # objectfiles
6
7  all: $(BIN)                 # Make all befehl
8
9  %.o: %.cpp                  # cpp Files zu o dateien compilen
10   $(CXX) $(CXXFLAGS) $<
11
12 $(BIN): $(OBJS)             # Linken zu exe file
13   $(CXX) $(LDFLAGS) $@ $^
14
15 run: $(BIN)                 # Projekt ausfuehren
16   ./$(BIN)
17
18 clean:                     # o. Dateien entfernen
19   rm -f $(OBJS) $(BIN)
20
21 .PHONY: clean all           # Markiert clean & all als PHONY
```

15 Styleguide

- **Variablen, Konstanten**
  - mit Kleinbuchstaben beginnen
  - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
  - keine Underscores

  Beispiele : counter, maxSpeed
- **Funktionen**
  - mit Kleinbuchstaben beginnen
  - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
  - Namen beschreiben Tätigkeiten
  - keine Underscores

  Beispiele : getCount(), init(), setMaxSpeed()
- **Klassen, Strukturen, Enums**
  - mit Grossbuchstaben beginnen
  - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
  - keine Underscores
  - Namen beschreiben Dinge

  Beispiele : MotorController, Queue, Color