C++

Programmstrucktur

Präprozessor

Er überarbeitet den Quellcode, sodass der Compiler daraus eine Objektdatei erstellen kann. Diese werden dann wiederum vom Linker zu einem Programm gebunden.

Da der Compiler immer nur eine Datei übersetzen kann müssen andere Dateien per *#include* eingebunden werden. Der Präprozessor läuft vor dem Compiler und verarbeitet alle Zeilen die mit einem Doppelkreuz (#) beginnen. Ein solcher Befehl kann auf mehrere Zeilen ausgedehnt werden, indem das Zeilenende mit einem vorangestellten Backslash (\) maskiert wird.

```
   #include "name"  // Sucht im aktuellen Verzeichnis und dann in den Standardpfaden des
                       Compilers
#include <name>  // Sucht gleich in den Standardpfaden des Compilers
```

Verwenden Sie für Verweise auf Ihre eigenen *Includes* immer eine relative Pfadangabe.

- "c:\users\manni\Eigene Dateien\code\cpp\projekt\zusatzlib\bla.h" schlecht!
- "../zusatzlib/bla.h" gut! Gesamtes Projekt lässt sich leichter in anderen Pfaden kompilieren.

`#define` belegt eine Textsubstitution mit dem angegebenen Wert, z.B.:

```
   #define BEGRUESSUNG "Hallo Welt!\n"
cout << BEGRUESSUNG;
```

Auch Macros möglich:

```
   #define ADD_DREI(x, y, z) x + y + z + 3
int d(1), e(20), f(4), p(0);
p = ADD_DREI(d, e, f);                  // p = d + e + f + 3;
```

```
   #defined ''Name''
```

Als Bedingungen sind nur *konstante Ausdrücke* erlaubt, d.h. solche, die der Präprozessor tatsächlich auswerten kann.

`#undef BEGRUESSUNG` `//` löscht eine definierte Präprozessorvariable bzw. ein Makro.

Verwenden Sie diese Makros vor allem als Zustandsspeicher für den Präprozessordurchlauf an sich und nicht um Funktionalität Ihres Programms zu erweitern.

Hier mehr zum Thema:
http://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Weitere_Grundelemente/_Vorarbeiter_des_Compilers

Header-Dateien:

Headerdateien sind gewöhnliche C++-Dateien, die im Normalfall Funktionsdeklarationen und Ähnliches enthalten.

Deklarationen machen dem Compiler bekannt, wie etwas benutzt wird.

Wenn Sie also eine Headerdatei einbinden und darin eine Funktionsdeklaration steht, dann weiß der Compiler wie die Funktion aufgerufen wird.

Der Code in der Datei die mit `#include` referenziert wird, wird vom Präprozessor einfach an der Stelle eingefügt, an der das `include` stand.

Da g++ ist sowohl Compiler als auch Linker, muss man mit dem Befehl einfach die zwei Dateien aufrufen und es wird automatisch ein fertiges Programm erzeugt.

Headerdateien haben oft die Endungen „.hpp“, „.hh“ und „.h“. Letztere ist allerdings auch die gebräuchlichste Dateiendung für C-Header, weshalb zugunsten besserer Differenzierung empfohlen wird, diese nicht zu benutzen.

Die Standardheader von C++ haben überhaupt keine Dateiendung (siehe: `iostream` und `string`). Historischbedingt akzeptieren die Compiler jedoch ''.h''.

Der Unterschied zwischen „iostream“ und „iostream.h“ besteht darin, dass in ersterer Headerdatei alle Deklarationen im Standardnamespace `std` vorgenommen werden.

Include guards

Wenn eine Headerdatei mehrmals ingebettet (#include <"bla.h">) verursacht es Problemme bei einigen Compilern. Die Lösung sind die include guards.

Es kann passieren, dass Headerdatei mehrfach eingebunden wird. Da viele Header nicht nur Deklarationen, sondern auch Definitionen enthalten, führt dies zu Compiler-Fehlermeldungen, da innerhalb einer Übersetzungseinheit ein Name stets nur genau einmal definiert werden darf (mehrfache Deklarationen, die keine Definitionen sind, sind jedoch erlaubt). Um dies zu vermeiden, wird der Präprozessor verwendet.

```
// Bla.h
#ifndef BLA_H  // Wenn die Datei BLA_N noch nicht definiert... (BLA_H = Makro)
#define BLA_H // Dann wird sie definiert (erzeugt)

Deklarationen und Definitionen der Headerdatei
...
#endif              //Wenn die Datei schon zum früheren Zeitpunkt definiert wurde, überspringt
                    der Präprozessor alle Deklarationen und geht sofort zum #endif.
```

Lösung Nr. 2

Spracherweitrerung *#pragma once*:

Diese sorgt ebenfalls dafür, dass eine (Header-)Datei nur einmal eingebunden wird, setzt jedoch auf höherer Ebene an (direkt am Präprozessor). Führt keine Makros ein.

Zur Verwendung genügt es, innerhalb der Header-Datei die Anweisung *#pragma once* einzufügen:

```
// A.h #pragma once

Definitionen
...
```

Vorsicht: wird nicht von allen Compilern unterstützt!

Include guards in jeder Headerdatei verwänden und man muss sich keine Sorgen mehr machen!

Compiler: (und Nützliches)

Das Schlüsselwort `inline` empfiehlt dem Compiler, beim Aufruf einer Funktion den Funktionsrumpf direkt durch den Funktionsaufruf zu ersetzen, wodurch bei kurzen Funktionen die Ausführungsgeschwindigkeit gesteigert werden kann. Es bewirkt aber noch mehr. Normalerweise darf eine Definition immer nur einmal gemacht werden. Da für das Inlinen einer Funktion aber die Definition bekannt sein muss, gibt es für Inline-Funktionen eine Ausnahme: Sie dürfen beliebig oft definiert werden, solange alle Definitionen identisch sind. Deshalb dürfen (und sollten) Inline-Funktionen in den Headerdateien definiert werden, ohne dass sich der Linker später über eine mehrfache Definition in verschiedenen Objektdateien beschweren wird.

Compiler compiliert die quelldataien und includiert die Header-Dateiel, als Resultat entstehen die Objektdateien (suffix = o, oder obj). Der Linker Verbindet diese Dateien zum Ausführbaren Datei (suffix z. B. exe)

//===

Flags sind optionen für den Compilator. Es könnte z.B. sein eine Bibliothek, die er einbinden muss, oder der Ort für die Header-Dateien:

- *gcc(g++) func_math* ***-c*** -> func_math.o (object=Maschinencode)
- *gcc(g++) func_math* ***-shared*** -> func_math.os (Bibliothek, in Windows .dll)
- *gcc(g++) func_math* -> wird nicht funktionieren, weil kein main

Die func_math ist ein Unterprogramm und hat kein main() um es trotzdem zu kompilieren muss man es dem compilator mit einem **flag -c** mitteilen, dass er eine Bibliothek erstellen soll.

Compilieren kann man zwei Programme auf folgendem Wege:

1. g++ main.cpp func_math.cpp // main + “c++”-Programm
2. g++ main.cpp func_math.o // main + Object (Maschinencode)
3. g++ main.cpp func_math.os // main + Bibliothek

- *gcc func_main* ***-L****/package/…/includes* ***-l****func_math.os (der name*

Mit dem flag **-l** teilt man dem Kompilator mit, wo die Header-Datei zu finden ist.

```
- G++ -o ~/directory/wunsch_name.exe name.cpp
                        // Kompielieren und den Namen gleich vergeben
```

Nützliches:

Generic: Ist die Eigenschaft eines materiellen oder abstrakten Objekts, die sich auf eine ganze Klasse, Gattung oder Menge anwenden lässt.

In der objektorientierten Programmierung werden Funktionen möglichst allgemein entworfen, um für unterschiedliche Datentypen und Datenstrukturen verwendet werden zu können.

Negative zahlen: Zweier-Kompliment: Das letzte Bit ist bei einer negativen zahl eine 1!
Zahl negieren: alle Bits negieren und 1 dazuzählen.

- #include <iostream> = Standartbib für Ein- und Ausgabe, deswegen in <> und nicht in " ".
- #include <time.h> = Standartbib für die Zeit- und Datumsoperationen . mehr siehe unter:
http://www.cplusplus.com/reference/ctime/

In Header-Datei warden die Deklarationen und in C die Definitionen geschrieben
- Shift+cmd+7 = Zeile auskommentieren
-Shift+cmd+0 = Hochgestellt
- Hungarian notaition = Bezeichnungswise von Variablen (z.B. kiVar = k=const/i=int)
const int kiVar = 7;
double = double precision;
Semikolon am Ende = Anweisung

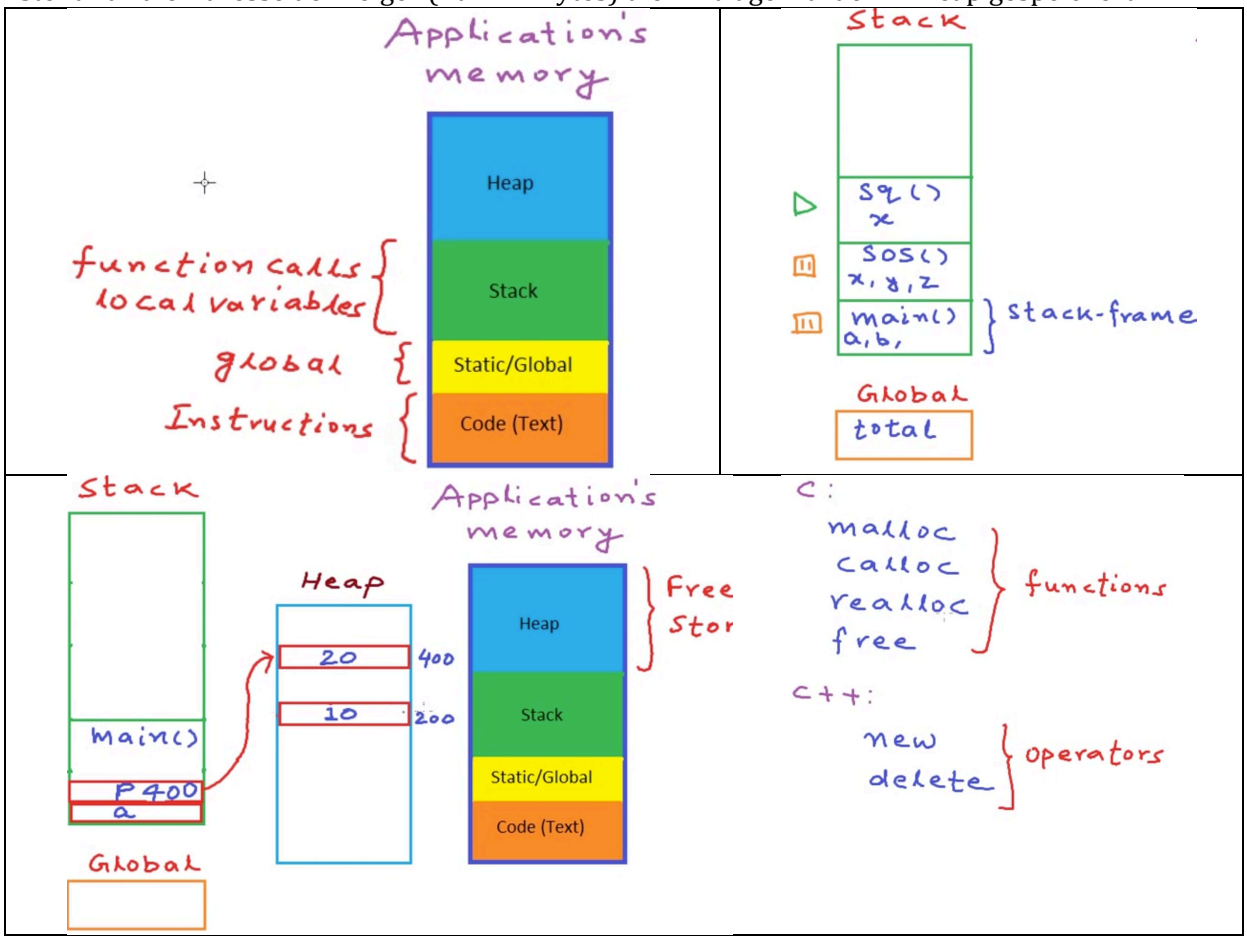
Sizeof(Klasse) = Größe eines Objektes der Klasse ohne der statischen Datenelemente und Methoden.
Int x = atoi(string); // konvertiert String in int

```
for (int i=0; i<x; i++)
    If(!(i%8))          // Jede 8 Schleifendurchgägne mache irgendwas.
```

Byte ist die kleinste adressierbare einheit im Speicher

Speicher:

RAM (Arbeitsspeicher): Lese und Schreibzugriffe des Programms
Heap-Speicher wird nur mit den Zeigern erreicht, die lokalen Variablen warden in Stack gespeichert, die Globalen- oder Staticvariablen und Constanten im separaten Speicher. Im Stack steht nur die Adresse der Zeiger (nur 2-4 Bytes) die Einträge warden in Heap gespeichert.



Stack is allocated during life time. It could happen that there are so many functions running, that the memory runs out this is called “stackoverflow”.

Alle lockalen Variablen warden im Stack gespeichert, um die Variablen im Heap zu hinterlegen wird der dynamisch Speicher verwendet. Nu Pointer und Arrays warden im Heap hinterlegt. Alle allocation Funktionen greifen auf Heap zu. Im Stack steht nur die Adresse der Variablen im Heap, je nach Systemarchitektur 2-4 Byte.

```
int main()
{
  int a; // goes on stack
  int *p;
  p = (int*)malloc(sizeof(int));
  *p = 10;
  free(p);
 p = (int*)malloc(20*sizeof(int));
}
```

P[0], P[1], P[2]

*p *(p+1)

```
int main()
{
   int a; // goes on stack
   int *p;
   p = new int;
   *p = 10;
   delete p;
   p = new int[20];
   delete[] p;
}
```

1. p = … Ein Speicher der Größe int wird im Heap reserviert.
2. *p = … der Wert 10 wird in den Heap geschrieben.
3. p =… Der Speicher in der Größe von 20 int’s wird im Heap reserviert.
4. Nicht vergessen den Speicher nach der Verwändung wieder freizugeben! free(p);/delete[] p;

EIN-und AUSgabe

Der Puffer für das Ein- und Auslesen ist für alle Funktionen gleich

Ausgabe

»setw« steht für »set width«, also »setze Breite«. Dieser wird vor der eigentlichen Ausgabe an **cout** gesandt und bereitet die Formatierung der folgenden Ausgabe vor. Zwischen die Klammern von `setw()` schreiben Sie die Anzahl der Stellen, die für die nachfolgende Ausgabe reserviert werden sollen. Alle Stellen, die nicht von der Zahl selbst belegt werden, werden mit Leerzeichen so aufgefüllt, dass die Zahl rechtsbündig erscheint.

```
cout << setw(7) << 3233 << setw(6) << 128 << endl;
```

Eingabe

```
int i, a, b, c;
cin >> i;                  // Eingabe in der int-Var i speichern
cin >> a >> b >> c;    // Drei Eingaben mit Tab, Enter, oder Leerzeile als Trenner
cin >> hex >> a oct >> b;          // Einlesen von Hex und Oct Zahlensystemen
                                   auch als 0x..., un 0... möglich
a = cin.get();          // cin.get-Funktionen lesen auch Zwischenraumzeichen ein
cin.get(a);             // analog zu cout.put(a)

// für char Variablen wird das '\0'-Zeichen automatisch hinzugefügt.
```

```
// Daten (Text) eingeben mit , trennen
// Programm ließt den text ein und gibt in bis zum Komma aus
int main(){
    char daten[64];

    int i=0;
    while ((daten[i] = cin.get()) != '\n') //String bis zum ENTER einlesen
        i++;
  daten[i]='\0';                    // '\n'-Zeichen (ENTER) mit '\0' überschreiben

    i=0;
  while (daten[i] !=',')           //String bis zum Komma ausgeben
        cout.put(daten[i++]);

    return 0;
}
// Besser Bibliotheksfunktionen getchar, putchar, da hie die Zwischenraumzeichen ingnoriert
werden, die eingabe jedoch mit ENTER beendet werden kann!

Cin und scanf beenden die Eingabe nach Zwischenraumzeichen.
```

Die Ein- und Ausgaben lassen sich auch mit den Funktionen ***scanf, printf*** durchführen.

```
scanf("%s", a);       // Liest String in a ein
printf("%s", a);      // Gibt String in a aus
```

%s bewirkt, dass Eingabe als Zeichenkette und a als Zeiger auf a interpretiert werden

```
scanf("%[^\n]", a);          // Liest String in a ein (nur das '\n'-Zeichen wird als
                             Abbruchsoperator definiert)
```

//***Gets*** Liest die Zeichenkette bis zum ENTER als einen String ein ('\0'). Die Adresse wird an das char-Array übergeben

```
char daten[64];
gets(daten);                 // Liest den String in daten bis zum ENTER ein (mit '\0')

char *z = daten;
gets(z);                     // Liest den String in den Zeiger z ein.

puts(daten);                 // ausgabe des char-Arrays
puts(z);                     // nach der ausgabe wird Zeilenvorschub durchgeführt
```

// Parameter: char-Array, wieviele Zeichen maximal eingelesen werden sollen, Delimiter.

```
cin.get(daten, 10, '\n')     // Der Delimiter wird nicht aus dem Puffer entfernt!
                             Beim nächsten aufrufen der Funktion wird nicht von der
                             Tastatur gelesen sondern der Delimiter in das char-Array
                             eingelesen.
cin.getline(daten, 10, '\n');        // '\n' per default als Delimiter eingestellt. Der
                             Puffer wird gelöscht.


cin.ignore(10, '\@')         // Lieset die Zeichen ein und löscht sie aus dem Puffer
cin.ignore();                // als default sind 1 Zeichen und EOF ('\n') eingestellt
```

Datentypen

Als einfache Regel zum Lesen von solchen komplexeren Datentypen können Sie sich merken:

- Es wird ausgehend vom Namen gelesen.
- Steht etwas rechts vom Namen, wird es ausgewertet.
- Steht rechts nichts mehr, wird der Teil auf der linken Seite ausgewertet.
- Mit Klammern kann die Reihenfolge geändert werden.

```
   int i;              // i ist ein int
int *j;             // j ist ein Zeiger auf int
int k[6];           // k ist ein Array von sechs Elementen des Typs int
int *l[6];          // l ist ein Array von sechs Elementen des Typs Zeiger auf int
int (*m)[6];        // m ist ein Zeiger auf ein Array von sechs Elementen des Typs int
int *(*&n)[6];      // n ist eine Referenz auf einen Zeiger auf ein Array von
                    // sechs Elementen des Typs Zeiger auf int
int *(*o[6])[5];    // o ist ein Array von sechs Elementen des Typs Zeiger auf ein
                    // Array von fünf Elementen des Typs Zeiger auf int
int **(*p[6])[5]; // p ist Array von sechs Elementen des Typs Zeiger auf ein Array
                    // von fünf Elementen des Typs Zeiger auf Zeiger auf int
```

- `bool` hat die Größe von min. 1 Byte (kleineste adressierbare Einheit), kann aber auch 4 Byte groß sein, da dies bei einigen Systemen die Geschwindigkeit erhöht.
- `char` ist der Standard-Datentyp für Zeichen und ist 1 Byte (256 Zeichen) groß was einen erweiterten (mit Umlauten) ASCII-Code darstellen kann.
  Jedes Zeichen ist einer Ganzzahl zugeordnet - ASCII 0-127.
  Die einzelnen Zeichen werden in **einfachen** Anführungszeichen gesetzt.

```
for(char i = 'A'; i <= 'Z'; ++i){     // Einfache Anführungszeichen ' ' !
  std::cout << i;
}
```

Ausgabe: **A**BCDEFGHIJKLMNOPQRSTUVWXY**Z**

- `wchar_t` ist ein Datentyp für Unicodezeichen und hat gewöhnlich eine Größe von 2 Byte.
- `Short`, `int`, `long` Ganzzahlen außer `wchar_t` können diese mit `signed` und `unsigned` verwendet werden.
- Laut ISO-C++ gibt es keine Größenvorgabe, nur: `char <= short <= int <= long`
- Außerdem ist festgelegt, dass `char` genau 1 Byte, `short` mindestens 2 Byte und `long` mindestens 4 Byte lang sein müssen.
- $2^{\text{Anzahl der Bits}}$, 1 Byte = 8 Bit

Typ	Speicherplatz	Wertebereich	kleinste positive Zahl	Genauigkeit
`float`	4 Byte	$\pm 3{,}4 \cdot 10^{38}$	$1{,}2 \cdot 10^{38}$	6 Stellen

`double`	8 Byte	$\pm 1{,}7 \cdot 10^{308}$	$2{,}3 \cdot 10^{308}$	12 Stellen
`long double`	10 Byte	$\pm 1{,}1 \cdot 10^{4932}$	$3{,}4 \cdot 10^{4932}$	18 Stellen

1. Deklaration: Dem Compiler die Variable bekanntgeben (Name, Rückgabewert)
2. Definition: Speicherplatz anlegen
3. Zuweisung: Wert der Variable verändern (x=1;). Bei `const` nur Initialisierung möglich.
4. Initialisierung: 1. Zuweisung (int x=1; int y(7)). Zustand der Variable/Objekts bei der Erstellung.

```
   int Zahl=100;  // Möglichkeit 1
int Zahl(100); // Möglichkeit 2

   int Zahl1(77), Zahl2, Zahl3=58;

   const int Zahl(400); // Alternativ: const int Zahl=400;
// oder
int const Zahl(400); // Alternativ: int const Zahl=400;
```

Explizite Typumwandlung

```
   static_cast< char > Variable
```

Gültigkeitsbereich

Gültigkeitsbereich der Variablen:

- Lokal: wird in der Funktion, oder in einem Block {} definiert und ist nur dort gültig
- Global: Außerhalb jeder Funktion am anfang des Programms gültig überall in diesem Programm
- Überdeckung: Sollten mehrere variablen den gleichen namen haben, wird die lokale Variable die Globale immer überdecken! Priotität: {} > f{} > global
- Zugriff auf die Globale variable die durch die lokale überdeckt wird erfolgt mit dem Bereichszugriffsoperator **::** => ::a;
    - Gilt nicht wenn die Variable innerhalb eines Blocks{} definiert wurde.

Um eine Variable auch außerhalb des Programms bekannt zu machen, muss die **Definition** (initiieren) mit dem Zusatz **extern** erfolgen. => **extern** int a =3;

Um auf eine Variable, die in einem anderen Programms definiert wurde zuzugreifen, muss die **Deklaration** ebenfalls mit dem Zusatz **extern** erfolgen. => **extern** int a;

Die deklaration dar niemals eine initialisierung enthalten!

Während die Definition einer globalen Variable auch ohne „extern" funktioniert ist der Zugriff auf eine Konstante hingegen ist ohne extern nicht möglich! Da diese nur in dem Programm wo sie definiert wurde bekannt ist.

Um Variablen und Funktionen in ihrem Wirkungsbereich zu beschränken wird dem extern entgegengesetztes **static** verwendet. Wird eine Funktion, oder eine Variable mit Static definiert ist sie nur innerhalb dieses Gültigkeitsbereichs verwendbar.

- static int a = 1;
- static void f(){}

Lebensdauer Variablen

auto (default) = Lebensdauer beträgt so lange wie der Block ausgeführt wird. Die Variablen werden im Stack (LIFO- Prinzp, Last In Frist Out) gespeichert und nach Beendigung der Anweisungen aus dem Stack entfernt.

C++ -Funktionen speichern im Stack die auto-Variablen und die Rücksprungadresse, wo Programm fortgesetzt wird, wenn die Funktion abgearbeitet ist.

Bei ineinander verschachtelten Funktionen droht ein **Stack Owerflow** da der Speicher so nicht leer gereumt wird. Jede Funktion bekommt einen Bereich im Stack zugewissen der sich über dem Bereich der der vorher aufgeruffenen und immer noch aktiven Funktion befindet.

Arrays:

http://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Weitere_Grundelemente/_Felder

Arrayvariable ist eine ZEIGERkonstante (Adresse) = kein modifizierbarer Wert!
Im Gegensatz zum Pointer kann für einen Array, also für alle Elemente eine Speichplatz allokiert werden. Rür Pointer nur für eine Adresse.

```
int  iaArray_1[3];       // Deklaration eines Arrays der Größe 3.
     iaArray_1[0]=1;     //Die Initialisierung fängt mit 0 an!
     iaArray_1[1]=2;
     iaArray_1[2]=3;     //Die Anzahl der Elemente ist drei, das letzte Element hat die Nr. 2!

   int iaArray_2[3] = {1,2,3};
int iaArray_3[ ] = {1,2,3};    // Flexiebele Anzahl der Elemente durch die Eingabe geregelt.
                               Die Arraygrüße kann jedoch im späteren verlauf nicht mehr geändert
       for(int i = 0; i < 3; ++i) {
           std::cout << iaArray_1[i] << ","; }   // Elemente 0 bis 9 ausgeben

    }
```

```
   int feld[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // 10 Elemente, mit Initalisierung

feld[2] = feld[1] - 5 * feld[7 + feld[1]];   // 2-5*feld(7+2) = 2-5*10
```

Vorsicht!: C++ überprüft nicht die Grenzen des Errays bei der Compilierung. Zugriffe über Arraygrenzen erzeugen undefiniertes Verhalten.

Arrays sind ähnlich einem konstanten Zeiger(Adresse) auf das erste Element. Es ist nicht möglich die Adresse des Feldes zu ändern. Der Unterschied zum Zeiger liegt daran, dass seine Größe nicht ein Element, sondern das ganze Array ist. (`sizeof()`)

```
std::cout << iaArray << std::endl;        Adresse des 1. Elements wird ausgegeben
```

```
Indexoperator:          feld[3]
Zeigerarithmetik:       *(feld+3)
```

Mehrdimensionales Array:

```
// Die Größe des Ersten Arrays muss nicht angegeben werden, sie wird
aus der restlichen größen und der {...} angabe errechnet.

int A[   ][ 3 ][ 4 ][ 2 ][ 5 ] = {};
      [n-1][n-2][n-3][n-4][n-x]  // Die Dimension des Arrays, n=5
       4D   3D   2D   1D   Element
```

```
Indexoperator:          feld[2][5]
Zeigerarithmetik:       *(*(feld + 2) + 5)
```

// Dereferenzierungen = Dimensionen nötig.

Ein char-Array kann dagegen komplett ausgegeben werden.

Die Eingabe im ersten Feld ist optional, da die Größe aus dem Zweiten feld plus der Elementeneingabe bestimmt werden kann.

```
int feld[optional][8] = { ... };
```

Für die Initialisierung ist jedoch besser:

```
int feld[2][3] = {{3,9,4},{,1,5,0}};
```

Die Fehlenden eingaben werden mit ‚0' initialisiert.

```
// Int-Größe wir vom Compiler bestimmt. Allgemein jedoch: (bei 16-bit-System: int=2Byte)
   Array Größe:  sizeof(feld)        // Arraygröße in Bytes (Arraygröße* int-Größe)
Element Größe:   sizeof(feld[0])
Element Anzahl:  sizeof(feld) / sizeof(feld[0])
   // Anzahl der Elemente eines Array = Arraygröße / Elementgröße
```

Mehrdimensionale Arrays sind Arrays vom Typ Array.

```
   // array[A][B] = {A {B}}

   int feld[3][8] = { // Mit Initialisierung
           {  1,  2,  3,  4,  5,  6,  7,  8 },
           {  9, 10, 11, 12, 13, 14, 15, 16 },
           { 17, 18, 19, 20, 21, 22, 23, 24 },
       };
   std::cout << "Größen\n";
   std::cout << "int[4][8]: " << sizeof(feld)   << "\n"; // 128 Bytes (int = 4 Byte)
   std::cout << "   int[8]: " << sizeof(*feld)  << "\n"; // 32=8 Elemente * (1 int)
   std::cout << "      int: " << sizeof(**feld) << "\n"; // 4 Byte

std::cout<<"     feld: "<< feld      <<"\n"; //Adresse des 1. Elem. von A(alle B)
std::cout<<"   feld[]: "<< feld[]    <<"\n";
     // Ist gleichzeitig die Adresse des Elementes [0][0], also des Ganzen Arrays:
std::cout << "(*feld+0)+0:" << (*feld+0) + 0 << "\n"; //Adresse [0][0]
std::cout<<" feld+1: "<< feld + 1  <<"\n";      // Adresse des 2. Elem. von A (alle B)
std::cout<<" feld [1]: "<< feld  [1] <<"\n";    //    - // -    A(1) = A(0)+ 8*4, B(X)
std::cout<<"(*feld) +1:"<<(*feld)+1 <<„\n“; // Adresse des 2. Elem. von B von A(0)
std::cout<<"feld[0][1]:"<< feld[0][1]<<"\n";    //    - // -    A(1) = A(0)+ 8*4, B(X)

std::cout << " Inhalt: "<< *(*(feld+1)+4)<< "\n"; // Inhalt((Inhalt von Adresse(1)+4))
   Gehe zu Adresse feld +1 dort steht eine weitere Adresse, gehe vod dieser +4
std::cout << " Inhalt: "<<    feld [1][4]<< "\n";
std::cout << " Inhalt: "<< *(*(feld+0)+12)<<"\n"; // Das Array ist nichts anderes als ein eindimensionaler Zeiger die Stelle 12 entspricht 2.Zeile, 4. Spalte = 8 + 4
```

feld **= Adresse von A(0)** Darin enthalten 8 weitere Adressen
(um zu A(1) zu gelangen muss man 8 B's überspringen = 8*4 Byte = 32d=20HEX)

***feld** **= Adresse B(0) von der Adresse A(0)** (Inhalt der Adresse ist wiederum eine Adresse)

Array	**B(0)**	**B(1)**	**B(2)**	**B(3)**	**B(4)**	**B(5)**	**(*feld+X)+6**	**B(7)**
A(0)	1	2	3	4	5	6	7	8
A(1)	9	10	11	12	(*(*feld+1)+4)	14	15	16

feld+2	17	18	19	20	21	22	23	24

Referenz/Alias:

Ist ein Alias einer Variablen, also dieselbe Variable, anderer Name, das heißt die zwei Variablen haben die **selbe Adresse**.

Referenz muss immer initialisiert sein und der Bezug kann nicht mehr geändert warden.

```
int  a = 10, b = 20;
int &r = a;        // Referenz r auf die Variable a

std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
      // Ausgabe:    10  20  10
++a;
std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
      //Ausgabe:    11  20  11
r = b;          // r (und somit a) wird 20 zugewiesen, r zeigt aber weiterhin auf a.
std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
            // Ausgabe:    20  20  20
```

Zeiger/ Pointer:

Pointer = Zeiger (Zeigervariable) in der Die Adresse einer „Variablen“ gespeichert ist.

Zeiger haben immer die gleiche Größe, egal auf welche Daten sie zeigen. Letztlich enthalten sie immer eine Speicheradresse, und die ist für alle Typen gleich. Die Größe ist von der Maschinenarchitektur abhängig. So ist auf den heutigen 32-Bit-Systemen ein Zeiger 32 Bit, also 4 Byte, groß. Die genaue Größe des Zeigers beträgt sizeof(int*).

Manchmal muss man eien Zeiger speichern, dessen Zieltyp sich erst noch im Laufe des Programms ergibt. Solche Zeiger werden als Zeiger auf den Datentyp `void` definiert.
Eine Variable vom Typ `void` gibt es nicht. Die Definition einer solchen Variablen würde also zu einem Fehler führen. Es ist aber durchaus erlaubt einen Zeiger auf `void` zu definieren.

‚&’ = holt die Adresse
‚*’ = Inhalt des Zeigers

- **void pointer** ist ein Zeiger dessen Datentyp noch nicht bekannt ist. Vorteil: Diesem Zeiger kann eine beliebige Adresse zugewiesen warden.

  Vorwiegend findet ein void-Zeiger Anwendung in Funktionen, anstatt für jeden Datentyp eine eigene Funktion zu schreiben, wird einfach der void-Zeiger verwendet, und der Funktion kann es egal sein, mit welchem Datentyp sie verwendet wird.

```
void pointer werden in c++ praktisch nie verwendet
(void *p = generic pointer type)
```

- Array sind **konstante** Zeiger! Zeiger kann man umdefinieren so, dass er auf andere Daten (Adresse) verweist, Array **nicht**!

- Zeiger verhalten sich oft wie eine Variable, z.B. lässt sich die Zeigervariable p = A (int *p = A) setzen, nicht jedoch andersrum, also: A = p;

Indizierte Zeiger in dereferenzierte Zeiger überführen:
Der Compiler wandelt stets Indexasdrücke in Zeigerausdrücke um.

```
int A [ I1 ][ I2 ][ I3 ][ In ] = {}; // *(A[I1][I2]...+In)
    *(*(*(*(A+I1)+I2)+I3)...+In)={};
```

```
Indexoperator:          feld[2][5]
Zeigerarithmetik:       *(*(feld + 2) + 5)
```

```
int main(){
    // & macht das Objekt zu einer Adresse
    // * Dereferenzierungsoperator macht die Adresse zum Objekt

int a = 7;        //  Variable definieren

int *b;           // Deklaration
```

```
b = &a;          // Definition (Adresse der Variable a in b speichern)

// Die Adresse wird immer in Hex angegeben!
// int *b = &a;    // Deklaration mit Definition

int **bb = &b;  // Adresse von p an bb zuweisen
int *pWert;
int Wert;

pWert = Wert;   // Dem Zeiger kann nur eine Adresse zugewiesen werden, kein Wert
Wert  = pWert;  // Eine Adresse kann keiner Variable übergeben werden.
Wert  = *pWert; // so funktioniert es!

b++;       // Gehe eine Adress weiter      +4 Byte
++bb;      // Gehe einen Zeiger weiter     +8 Byte

//******************************************************

int s = *b;     // In der Variablen s den ersten Arraywert speichern
int s = b[0];   // das Gleiche!  Zeiger ≈ Array

int c[] = {7,11,15};       // Array anlegen
int *d = &c[2];       // Zeiger auf die Dritte(0,1,2..) Stelle im im Array c
int *d = &c[];        // geht nicht!
int *e =c;            // *e = e[], c[] = *c
//Der Name eines Arrays ist die Adresse des vordersten Eintrags = ein Zeiger!
//Also weise dem Zeiger "c" den Zeiger "e" zu!

*d = 6;             // Die dritte Stelle(15) mit 6 überschreibe

e++;            // Die Adresse wird um "eins" (eine Adresse) erhöht
                //Weil c int(16-bit-System) ist e um 2 Byte erhöhen.
                // Der Zeiger e verändert sich jedoch nicht.

e[0] = 7;       // In die erste Position des Zeiger 3 reinschreiben,
                also da vo vorher die 11 wahr.

e[1] = 13,7;     // Da der Zeiger als Int declariert wird wird auch
                int reingeschrieben: 13

*(e+1) = 2147483648 ;   // Vorsicht wenn zu lang wird von long in
                int convertiert (+1 dazuaddieren)

// Überlauf: Zählt bis zur der höhsten Zahl von int (z.B. 32xxx bei 16-Bit-System) macht einen
loop und fängt wieder an bei der kleinsten Zahl an (-32xxx)  => (z.B. 8-Bit-System: (signed)
1111.1111=511 -> 512=1.0000.0000 => 0000.0000 also – 511

//*****************************************************

// const gehört immer zu dem was links von ihm steht
// Es sei denn links steht nichts mehr, dann gehört es zu dem was
rechts davon steht.

int        Wert1;              // eine int-Variable
int        Wert2;              // noch eine int-Variable
```

```
int const *p1Wert = &Wert1;      // Zeiger auf konstanten int (Der
                                 Inhalt des Zeigers (Wert) kann nicht
                                 geändert werden)
int * const p2Wert = &Wert1;     // konstanter Zeiger auf int (Zeiger
                                 kann nicht geändert werden)
                                 // Arrays sind konstante Zeiger!

int const * const p3Wert = &Wert1;    // konstanter Zeiger auf
                                      konstanten int

p1Wert  = &Wert2; // geht (Zeiger wird „umgebogen“)
*p1Wert = Wert2;  // geht nicht, der Wert ist konstant

p2Wert  = &Wert2; // geht nicht, Zeiger konstant
*p2Wert = Wert2;  // geht, die Werte des Zeigers sind eränderbar

p3Wert  = &Wert2; // geht nicht, Zeiger ist konstant
*p3Wert = Wert2;  // geht nicht, der Wert ist konstant

//*********************************************************
```

Zeigerarrays:

```
const int *(z[10]);   // Array von 10 Zeigern auf int-Objekte
int i=1;
z[0] = &i;            // der Erste Zeiger verweist auf i

//******************Eine Liste aus Stings*******************
char Zahlen[3][5]= {"Eins", "Zwei", "Drei"};
// Diese Variante ist schlecht, weil mann im Voraus die
Buchstabenzahl (5) festlegen muss und diese dann konstant für alle
Einträge bleibt.
char *(m[]) = {"Eins","Zwei", "Drei"};    // Funktioniert seit C++11 nicht mehr
const char *(m[]) = {"Eins","Zwei", "Drei"}; // nur als const String! Kann
                                             jedoch dann nicht mehr geändert werden!
// Die beste Methode ist:      Zeiger auf Zeiger auf char

// int **zz = *z = (&z = i) = Element!
// *zz = z = &i = Adresse des Elements!

// *(*(zz+2)+1) = 2.Zeichen des 3. Elements
//   *(zz+2)+1) = Adresse des 2. Zeichens des 3. Elements
```

Es wird immer Platz für einen String (Zeiger auf char) allokiert

Bei jeder eingabe wird der Speicherbereich um `1*sizeof(Zeigerdatentyp)` vergrößert.

```
// In zwei Schritten:
char a[]= "Eins", b[]= "Zwei", c[]= "Drei";     //Strings definieren
char *z[] ={a,b,c};        // Array mit den Strings initialisieren

z[2][0]='F';          // Nun können die Strings auch verändert werden
```

```
cout << "z_neu: " << z[2] << endl;     // Statt „Drei“ wird „Frei“ ausgegeben
```

Dynamische Speicherverwaltung:

Zeiger warden beim Dynamischen Speicherkonzept benutzt. Beim Programmieren weiß der Programmierer noch nicht wieviel Speicher er für eine Variable benötigt.

```
int main(){

int n;
cin >> n;
short array[n];          // Muss nicht zwangsleufig zum Fehler beim Kompilieren führen
                         verursacht jedoch Probleme, da der Compiler bereits zum
                         Übersetzungszeitpunkt  des Programms den Speicherplatz
                         reserviert.
short array1[n]={};    // Fehler

//*****************************************************

int *z;                        // Zeiger auf ein int-Wert definieren
z = new (int);           // Speicherplatz für einen int-Wert allozieren
                         und die Adresse dem Zeiger z zuweisen.

int *z = new (int);          // Deklaration mit Definition
z = (int*) malloc(sizeof(int)); // Das gleiche für C
z = new (int)(7);      //So wird der Zeiger initiiert, sonst ist der
                         Inhalt zufällig!


delete z;          // Des Speicher wieder freigeben
                   // Mann kann den zeiger nicht zwei mal löschen!
z = NULL;          //Zur verdeutlichung, dass der Speicherplatz nicht
                   mehr zur verfügung steht und um Fehler zu vermeiden
                   den Zeiger umbrauchbar machen.

//=====================ARRAY's=========================

z = new int[10];         // Ein Array der Größe 10 (oder auch eine
                         Variable) allozieren
z = (int*) malloc(10*sizeof(int)); // Das gleiche für C

z = new (int)[6]={2,6,5,8,7,4}; // So nicht möglich!!!
for (int i=0; i<6; i++)          // Initialisierung, so geht's!
    z[i]=0;

delete []z;      // Speicher des Arrays freigeben (nur für new)

// Solte nicht genügend Speicher forhanden sein wird nicht die
Adresse sonder der NULL-Zeiger zurückgegeben!!!
```

```
// Man sollte eine Error-Funktion einbauen!

//*****************************************************

int n = cin.get();
short *array = new short[n];     // Zur Laufzeit ein Array der Größe n allokieren

// Mehrdimensionales Array

int n=cin.get();
int (*z)[4] = new int[n][4]; // Zeiger auf 4-elementige Arrays
int  z[][4] das Gleiche, jedoch umständlicher zu Arbeiten
//Klammern () sind notwendig, da Priorität von [] > *
int *z[];  //4-elementiges Array aus Zeigern die auf int-Werte zeigen

z[0][0] = 3;                // Zugriff auf die Daten


//************ malloc, calloc realloc auch für C ************

#include <stdlib.h>

void *p = malloc (4);       //malloc reserviert einen Block
                            Speicher, in diesem Fall 4 Bytes
                            (void *p = generic pointer type)

// alle diese Funktionen liefern einen generischen Zeiger des Typs
void* zurück.
Da ein beliebiger Zeiger einem voir-Zeiger zugewiesen werden kann,
jedoch nicht umgekehrt, muss man den void-Zeiger konvertieren.

int *i;
void *z;
    z = i;      // das geht
    i = z;      // das nicht!
i = (int*)z;    // explizite Typen Konvertierung

int *t = (int*) malloc(n * sizeof(int));
//reserviert Speicherplatz für n Integer und Speichert die Adresse
des Ersten in dem Zeiger *t

//Funktioniert genauso mit mehrdimensionalen Arrays!
int *namen = (char**) malloc(names * sizeof(char*));
// Funktioniert genau so für calloc und realloc.

int *z = (int*) calloc(n, sizeof(int));
//Argumente:    Anzahl der Elemente
                Größe in Bytes
 Außerdem werden alle Elemente mit dem Wert 0 initialisiert.

int *y = (int*) realloc(z, n*sizeof(int));
// Größe des allokierten Speichers nachträglich modifizieren!
Wenn man schon einen Speicherblock alloziert hat und diesen nun ändern muss, z.B. größer
machen. Der Zeiger wird umbebogen, wenn de alte Block nicht vergrößert werden kann.
```

```
Argumente:      Z = der zu Verändernde Zeiger.
                Sizte_t = neue Größe des zu allozierenden Speichers

// Immer auf Fehler in der Speicherplatzreservierung  achten!

char **z;
char **backup;
backup = z;     //Den zeiger immer Sichern, das im er im Falle eines
                Allokationfehlers gelöscht wird, z = NULL;

if(z ==NULL){
        cout << „\nFehler bei der Allokation!“
        // --- weitere Anweisungen---
    }

free(y);        //Speicher freigeben
```

```
/(****************** Liste aus Strings II - Dynamische Strings ********************************

int main(){
    char **namen = NULL;// Zeiger auf einen dynamischen Array ( init. Null-Pointer)
    char **backup = nullptr;     // Dicherungskopie
    char buffer [128];           // Einlesespeicher
    int scnt = 0;                // Anzahl Einträge (String CouNTer

do{
    cin.getline(buffer, 128);

     if(strcmp(buffer, "---"))      // soll "---" Ende der eingabe signalisieren
          backup=namen;

// Speicher platz allokieren und nach dem Fehler abfragen
1. Speicher für Array von Strings (Anzahl der eingelesenen String
    + 1 für den nächsten)
2. Speicher für den eingelesenen String: Buchstabenanzahl+1 für '\n'


     if ((namen = (char**) realloc(namen, (scnt+1)*sizeof(char*)))
     ||
     (namen[scnt] = (char*) malloc(strlen(buffer)+1)) == NULL){

          namen = backup;  // namen wiederherstellen, sonst namen = NULL
     }

    strcpy(namen[scnt], buffer);// String im Zeigerarray(namen[scnt]) abspeichern
           scnt++;              // Anzahl für strings um 1 erhöhen

}while (strcmp(buffer, "---")); // Wenn Eingabe beendet ("---") do stopen.
```

Mit den Operatoren new und malloc allokiert der Compiler den Speicherplatz im Heap (Bereich des Arbeitsspeichers) erst zur Programmlaufzeit und liefert die Anfangsadresse des Speicherblocks zurück. Mit delete oder free, wird der Speicher wieder Freigegeben.

Zeichenketten:

Um in C eine Zeichenkette (ein Wort, oder Satz, …) zu implementieren muss man ein Array von Zeichen anlegen.

String = char-Array das mit `'\0'` abgeschlossen wird.

```
#include <string>              // für die String-Funktionen wie std::string()

    char a[] = {'H','a','l','l','o','\0'};        // char-Array mit 12 Elementen
    int  b[] = {'H','a','l','l','o','\0'};         // int-Array mit 12 Elementen
    std::string z = {'H','a','l','l','o','\0'}; // char-Array mit 12 Elementen

einfacher:    char a[]   = "String";   // '\0' wird automatisch angefügt!
besser:       char *a    = "String"; //Im Gegensatz zum Array Zuweisbar!

Besonderheit der Stringkonstante (char):
cout << a;        // Es wird nicht die Adresse ausgegeben, sondern der String!!!
cout << &a;       // Adresse der Stringkonstanten

int b[] = {1,2,3};
cout << b;        // wird die Adresse des b-Zeiger ausgegeben.
cout << *b;       // Wird der Inhalt des ersten Arrayelementes ausgegeben: 1
```

In C++ gibt es dafür eine Klasse namenst `string` in der Standardbibliothek.

Während einzelne Zeichen in einfachen Anführungszeichen gesetzt werden braucht man die doppelten Anführungszeichen um eine Instanz eines `char`-Arrays zu erzeugen.

```
#include <string>                          // Einbinden der Standardbib.

  char a[] = "Hallo Welt!";          // char-Array mit 12 Elementen
  std::string z = "Hallo Welt!";          // char-Array mit 12 Elementen
  std::cout << zeichenkette << std::endl;      // Ein- und Ausgabe des string-
  std::cin  << zeichenkette << std::endl;      // Objekts ist problemlos möglich
```

Der Konstruktor für das `string`-Objekt `z` wir aufgerufen und erhält als Parameter das `char`-Array `"Hallo Welt!"` welches es in ein String aus einzelnen Zeichen umwandelt (siehe oben).

Wie bereits bekannt, können an Funktionen keine Arrays übergeben werden. Stattdessen wird natürlich ein Zeiger vom Arrayelementtyp (also `char`) übergeben.

Dabei geht aber die Information verloren, wie viele Elemente dieses Array enthält. Jedoch kann anhand des '\0'-Zeichen (Nullzeichen) auch innerhalb der Zeichenkette erkannt werden, wie lang die übergebene Zeichenkette ist.

Die Zeichenkette wird nur bis zur nächsten Leerzeiche eingelesen.

```
std::getline(std::cin, zeichenkette);              //Liest bis zum Zeilenende
```

Oder man kann einen dritten Parameter kann man das Zeichen angeben, bis zu dem man einlesen möchte, z.B. `'y'`.

```
std::getline(std::cin, zeichenkette, 'y');    // Liest bis zum nächsten y
```

Zuweisen und Verketten:
Verkettung: **+-** Operator
Anhängen : **+=** -Operator
Was drin steht + vas zugewiesen wird.

```
int main() {
 std::string string1, string2, string3;

 string1 = "ich bin ";
 string2 = "doof";
 string3 = string1 + string2;
 std::cout << string3 << std::endl;        // ich bin doof
 string3 += " - " + string1 + "schön";
    std::cout << string3 << std::endl;            // ich bin doof - ich bin
schön

 std::cout << string1 + "schön " + string2 << std::endl;
  // Ich bin schön doof
}
```

```
«stringname».«methodenname»(«parameter...»);
```

- size() /length() // Länge der Zeichenkette
- empty() // Leer = true, nicht leer = false
- clear() // String leeren.
- resize() // Zwei Parameter. 1. neue Größe
  2. Womit wird der String gefüllt (default = 0)
- swap() // Den Inhalt zweier Strings auszutauschen. (Param = 2. String)
- find() // Sucht nach einem String und gibt den Index zurück,
  der 2. Param gibt die Startposition für das Suchen an.
  `suche="xy"; string.find(suche, 0);`
- erase() // string.erase(5, 1) Startwert, Anzahl der Zeichen/ leer = alle
- replace() // Anfangsposition und die Anzahl der Zeichen, die ersetzt werden.
- insert() // Einen String an einer bestimmten einfügen.
- substr() // Substring zurückgeben: Startwert, Länge

```
       std::string string = "Ich bin ganz lang!";

       std::cout << string[4] << std::endl;
       std::cout << string.at(4) << std::endl;


       std::cout << string[0] << std::endl;     // Ausgabe von Datenmüll
       //std::cout << string.at(20) << std::endl;     // Laufzeitfehler wegen
überlauf

       std::cout << string.size() << endl;
       std::cout << string.find("a", 10) << endl;
       std::cout << string.erase(13,3) << endl;
       std::cout << string.empty() << endl;

       std::string str_2= "Zeichenkette";
       str_2.replace(str_2.find("k"), std::string("kette").length(),
"test");
    //                  Startpos,          Länge,              Ersatz
       std::cout << str_2 << std::endl;

       std::string str = "Hallo Welt.";
```

```
        str.insert(5, " schöne");
        std::cout << str << std::endl;
        std::cout << str.substr(0, str.find(' ') - 0) << std::endl;
     // „- 0“ = der Startwert muss abgezogen werden

        string.resize(7);
        std::cout << string << endl;
```

Strings können auch einfach miteinander verglichen werden.
Bei == muss der String exakt übereinstimmen. > / < / => / =< - Vergleich ergibt sich daraus, dass jeder String einer Zahl zugeordnet ist, siehe ASCII-Tabellen.

Zahlen und Strings umwandeln:

- Die C-Funktionen `atof()`, `atoi()`, `atol()` und `sprintf()`
- C++-String-Streams `std::ostringstream, std::istringstream`

Stringstreams funktionieren im Grunde genau wie die Ihnen bereits bekannten Ein-/Ausgabestreams `cin` und `cout` mit dem Unterschied, dass sie ein `string`-Objekt als Ziel benutzen.

```
#include <sstream>   // String-Ein-/Ausgabe
// ==== ZAHL_ZU_STRING=========
std::ostringstream strOUT;      // Unser Ausgabe-Stream ZAHL_ZU_STRING
std::string zeichen;
int var = 10;

strOUT << var;                  // ganzzahlige Variable auf Ausgabe-Stream ausgeben
str = strOUT.str();             // Streaminhalt an String-Variable zuweisen

std::cout << str << std::endl;  // String ausgeben
 // ====STRING_ZU_ZAHL=========
std::istringstream strIN;       // Unser Eingabe-Stream
str = "17";                     // Ein String-Objekt

strIN.str(zeichen);                     // Streaminhalt mit String-Variable füllen
strIN >> var;                   // ganzzahlige Variable von Eingabe-Stream einlesen

std::cout << var << std::endl;  // Zahl ausgeben
```

Zahl -> stringsteam -> String_var.

String -> stringsteam -> Int_var

Streaminhalt: streamingvar.str(); Streaminhalt mit Stringvar. füllen:

streamingvar.str(zeichen)

Statt istringstream und ostringstream können Sie auch ein stringstream-Objekt verwenden

Zeichenzugriff:

Es gibt zwei Arten auf einzelne Strings innerhalb der Zeichenkette zuzugreifen:

1. [ ] Wie bei Arrays lässt sich mittels des Indexes auf das gewünschte Zeichen zugreifen. Dabei weid keine Grenzprüfung durchgeführt.

2. at.( ) Mit diesem Operator wird auch geprüft ob der Wert innerhalb der Grenzen liegt. Im Fehlerfall löst sie eine `out_of_range`-Exception aus.

```
   cout << zeichenkette.length() << endl;              // 18

   std::cout << zeichenkette[4] << std::endl;
   std::cout << zeichenkette[20] << std::endl;         // Ausgabe von
Datenmüll
std::cout << zeichenkette.at(20) << std::endl;    // Laufzeitfehler
```

Zuweisungsoperatoren

a) Kopieren

```
char a[11], b[11]="Ein String"; // Zeiger auf den Datentyp char
strcpy(a, b);                 // Kopiert den String b in den Strin a, jedoch ohne '\0'
strcpy(a, "Ein String");          // dabei muss a mindestens so groß wie b sein
strcpy(a, &b[4]);   // Kopiert den String b ab dem fünften Arraelemet, also nur „String“
strcpy(a, b+4);
```

stpcpy ist der strcpy-Funktion gleich, liefert jedoch als Rückgabewert die Adresse des letzten zeichen.

b) Verknüpfung (eng. concatenation)

```
char a[50] = "Strings", b[15]="zusammenfügen"; // Die Größe des ersten
                                                  Strings beachten!
strcat(a, b);                              //Zusammenfügen fon zwei Strings
strcat(a, " zusammenfügen");
```

c) Strings vergleichen

```
strcmp(a, b);    // Vergleicht die Strings Elementenweise
                 // Erg = 0: Strings identisch
                 // Erg = negative Zahl: das erste Element von S1 besitzt einen Code-
                 Wert der Nach der ASCII-Tabelle größer ist als S2
                 // Erg = negative Zahl: Der Code-Wert ist kleiner
```

d) Länge(Anzahl der Zeichen) eines Strings

```
char a[] = "String", b[]="zusammenfügen";

strlen(„String“);      // Die Länge beträgt 6
strlen(b);             // Vorsicht bei Umlauten hier ist die Länge nicht 13, sondern 14!
```

e) Nach einem Zeichen im String suchen.

```
char a[] = "String", b='i';
    strchr(a, b);      // Gibt die Adresse wo sich das Zeichen ‚i' befindet.
    strchr(a, 'i');    //Vorsicht!!! Das Zeichen muss im Array vorhanden sein!
```

Bei Ausgabe wird wie gewönhlich das ganze Array bis zum ‚\o'-Zeichen ausgegeben, hier

also: „ing“

f) String Dublizieren

```
char a[] = "String", *c;

c=strdup(a);      // Kopiert/Dubliziert einen String in einen neuen Speicherbereich.
cout<<"a:"<<a<<"c:"<<c<<endl;     // Da die Arrays vom Typ char sind werden sie
                                    Als ganzes ausgegeben.
cout<<"a: "<<*a<<"  c: "<<*c;     // Hier wird jeweils nur die das ausgegeben, was
                                    in der Adresse steht, also nur ein Zeichen: 'S'
/* strdup bestimmt zuerst die Länge des übergebenen String s, reserviert danach mit malloc
einen Speicherbereich entsprechender Größe (strlen(s) + 1) und kopiert den Inhalt von s in
diesen neu reservierten Speicherbereich. Der Programmierer muß den Speicher selbst
wieder freigeben, wenn dieser nicht mehr benötigt wird. */
```

Es gibt noch zahlreiche andere, z. B.: (http://www2.informatik.uni-halle.de/lehre/c/c_fctstr.html)

{...} Mit den Klammern wird ein Bereich gekennzeichnet, nicht nur für Funktionen und Anweisungen, sondern auch für Variablen – Gültigkeitsbereich.

Namespace's können an jeder Stelle – außerhalb von Funktions- und Klassendefinitionen – geöffnet und geschlossen werden und somit in beliebig vielen Quelldateien erweitert werden.

```
#include <iostream>

using std::cout;                    //Namensraum std der C++-Standardbibliothek
using std::endl;

using namespace std;            // macht alles aus dem Namensraum std bekannt

namespace LongNameNameSpace{
   void f(){
      cout << "Long Name Name Space \n";
   }
   namespace NestedNameSpace{
      void g(){
         cout << "Nested Name Space \n";
      }
   }
}

int main(int argc, const char * argv[])
{
   namespace LNNS  = LongNameNameSpace;
   namespace NNS   = LNNS::NestedNameSpace;

   LNNS::f();
   NNS::g();
}
```

Da durch den Gebrauch von `using namespace` Namensbereiche ihren ursprünglichen Sinn, der Schutz vor Merfachbenennung, verlieren, wird es als besserer Programmierstil angesehen, wenn man die Elemente einzeln mit der `using-deklaration` einbindet.

--

:: = Zugrifs/ Bereichsoperator um auf eine Methode einer Klasse zuzugreifen.
Klasse :: Methode
Methoden sind dann eigentlich immer die Deklarationen der Funktionen zu den Elementen einer Klasse: Funktion

Um die eigentliche Funktion zu programmieren schreibt man dann mit:
Klasse::Methode()
{…

Die Konstruktoren der abgeleiteten Klassen initialisieren alle Basisklassen. Du kannst hier etwas an deine Konstruktoren einer Basisklasse übergeben. Tust du das nicht explizit, so wird implizit der Standardkonstruktor der Basisklasse aufgerufen.
MyClass::MyClass(int val, bool val2) : Base1(val), Base2(val2) {}
Hier initialisieren ich die Basisklassen explizit.

MyClass::MyClass() {}
Hier werden jeweils implizit die Standardkonstruktoren meiner beiden Basisklassen aufgerufen.

- `std::cout` Alle Funktionen, Klassen, Objekte der Standard-Bibiothek sind im Namensbereich ***std*** definiert.
- `::globaler_Name` Der Bereichsauflösungsoperator vor ***::globaler_name*** erlaubt auf den globalen Namen zuzugreifen selbst dann, wenn er durch einen lokalen gleichnamigen Bezeichner verdeckt ist.

- **::**iX: mit zwei „colons“ Wird auf die globale Variable zugegriffen, auch wenn ein lokale Variable den selben Namen hat.
- **<<** = *Ausgabeoperator (insertion operator)* , ermöglicht beliebig viele Teile der Ausgabe aneinanderzuhängen, die sich auch auf mehrere Zeilen verteilen dürfen:

```
cout << "Hallo, " << "Ihr" << " " << 2 << "!" << endl;
std::cout << std::endl;              // Leerzeile mittels endl
```

bewirkt die Ausgabe: `Hallo, Ihr 2!`

http://de.wikibooks.org/wiki/C++-Programmierung:_Einfache_Ein-_und_Ausgabe

- **>>** = extraction perator Eingabeoperator std::cin >> iNumber

```
int iAge;
std::cout << "How old are you?" << std::endl; //Ausgabe
std::cin >> iAge;                            //Eingabe
std::cout << "You are " << iAge << " years old." << std::endl;
```

- *\n* und *std::endl* (end line) Zeilenabschluss

"*\n*" und der Manipulator "*::std::endl*" bewirken die Ausgabe eines Neuzeilenzeichens.

::std::endl bewirkt zusätzlich danach eine„Synchronisation“ **:** Alle vorübergehend noch zwischengespeicherten Daten werden dabei tatsächlich ausgegeben.

Vorteil: Falls ein Programm durch einen Fehler verfrüht abgebrochen wird, dann sind alle so abgeschlossenen Zeilen nach dem Abbruch sichtbar (hilfreich bei der Fehlresuche).

Nachteil: Langsammer

\n sollte immer dann bevorzugt während ohne Synchronisation möglicherweise dann nicht alle ausgegebenen Informationen auch sichtbar werden.verwendet werden, wenn eine Synchronisation gar nicht nötig ist oder ohnehin erfolgt, wie z.B. am Programmende (schneller).

alls möglich, zu einem sowieso vorhandenen Textliteral hinzuzufügen, wie bei dem Textliteral *"Hallo, Welt!\n".*

```
   int main()
{ ::std::cout << "Hallo, Welt!" << ::std::endl;       am langsamsten!
  ::std::cout << "Hallo, Welt!" << "\n";
  ::std::cout << "Hallo, Welt!" << '\n';
  ::std::cout << "Hallo, Welt!\n"; }                  am schnellsten!
```

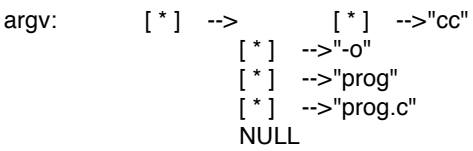
---

argc und **argv** **= arg**ument **c**ount and **v**ector

argc = Anzahl von Argumenten, die dem Programm beim Start übergeben wurden.

argv = Dabei handelt es sich um einen Integerwert. Im zweiten Parameter stehen die einzelnen Argumente. Diese werden als Strings in einer Stringtabelle gespeichert.

```
int main(int argc, const char * argv[ ]);        int main(int argc, char **argv);
```



Operatoren:

Ist im eigentlichen Sinne auch eine Funktion, wird nur anders aufgerufen.

Basic Input/Output:

stream	description
cin	standard input stream
cout	standard output stream
cerr	standard error (output) stream
clog	standard logging (output) stream

Zeichen	Umwandlung
%d oder %i	int
%c	einzelnes Zeichen
%e oder %E	double im Format [-]d.ddd e±dd bzw. [-]d.ddd E±dd
%f	double im Format [-]ddd.ddd
%o	int als Oktalzahl ausgeben
%p	die Adresse eines Pointers
%s	Zeichenkette ausgeben
%u	unsigned int
%x oder %X	int als Hexadezimalzahl ausgeben
%%	Prozentzeichen

'\n' = Leerzeile '\t' = einen Tab einrücken

%d = Ausgabe eines Integer-Wertes als Dezimalzahl. = c++?

```
Zahl  = 22;
Zahl += 5; // Zahl = Zahl + 5;
Zahl -= 7; // Zahl = Zahl - 7;
Zahl *= 2; // Zahl = Zahl * 2;
Zahl /= 4; // Zahl = Zahl / 4;
Zahl *= 3 + 4; // Zahl = Zahl * (3 + 4);
```

Unterschied zwischen posfix (i++) und prefix (++i) besteht im Rückgabewert.

```
int i=5;
cout<<"i = " << i << endl;                    // => i = 5
cout<<"i++ = " << i++ << endl;       // => i = 5 (Ausgabe des Wertes vor ++)
i=5;                                                    // i auf 5 zurücksetzen
cout<<"++i = " << ++i << endl;       // => i = 6 (++ - Wert wird zurückgegeben)
int max = (a>b) ? a : b;        // if-else-Konstrukt (? = if/true;   : = else/false)
```

Anweisungen:

if

- `if (condition) {...}`

```
   min = a < b ? a : b; // condition? "true!" : "false!"
// Alternativ ginge:

if (a < b) {
    min = a;
} else {
    min = b;
}
```

- `for (int i=0, j=4; i<4; i++, j++) {...}`

while:

kopfgesteuerte Schleife handelt, wird **erst die Bedingung ausgewertet**. Ist diese erfüllt, so wird die Anweisung ausgeführt ...

```
int i = 0;
while(i < 10){          // Erst wird die Bedingung ausgewertet!
    i++;
    std::cout << i << std::endl;
}
```

do while:

Gegenüber while um einen Durchgang schneller, weil die Bedingung am Ende nach "++"überprüft wird. Auch wenn die Bedingung unwahr ist, wird die Schleife auf jeden Fall mindestens ein Mal ausgeführt!

```
int i = 0;
do {                        // Die Schleife wird min. ein Mal ausgeführt!
        i++;
        std::cout << i << std::endl;
} while(i < 10);            // Erst am Ende wird die Bedingung ausgewertet
```

Beispiel: Fortsetzung einer Aktion

```
char weiter;
do {
        std::cout << "Schleife fortsetzen? (y)" << std::endl;
        std::cin >> weiter;
  } while(weiter == 'y');       //Beachte '' einsetzten nicht " "
```

For-Schleife:

Sie können also problemlos 2`int`-Variablen anlegen, aber nicht eine `int`- und eine `char`-Variable

```
unsigned int benutzer;                  // Variablen für Benutzereingabe
cin >> benutzer;                        // Benutzer gibt Zahl ein

for(unsigned int i = 1; i <= benutzer; ++i, Anweisung 2, ...)
```

```
      //Die Variable wird nur beim ersten Durchlauf initialisiert
      cout << i << endl;                        // Ausgabe von i
```

Break-Anweisung:

Sinnvoll in Verbindung mit einer Endlosschleife.

```
   unsigned int benutzer, i=1;     // Variablen für Benutzereingabe
   cin >> benutzer;                // Benutzer gibt Zahl ein

   for(;;){                        // Endlos-for-Schleife
     cout << i << "\n";          // Ausgabe von i
     ++i;                        // Variable erhöhen
     if(i>benutzer)break;        // Abbrechen, wenn Bedingung erfüllt
   }
```

Continue-Bedingung:

```
   for(unsigned int i = 1; i <= benutzer; ++i) {    // for-Schleife
      if (i%2 == 0) continue;      // Der Rest (geraden Zahlen) wird überspringen
      cout << i << endl;           // wenn i = 2 wir diese Zeile nicht ausgeführt
}
```

Switch and Break:

```
   switch(«Ganzzahlige Variable»){
      case  1:    «Anweisungsblock»
      case  2:    cout << "z.B. einen Text ausgeben";
      «...»
      default:    Anweisung falls nichts zutrifft
}
```

Forsicht! Es **wird alles ab dem gewählten ausgegeben**! Dies kann mit break verhindert warden:

```
   switch(«Ganzzahlige Variable»){
      case  1:    «Anweisungsblock»
      break;      //Bricht beim Zutreffen hier ab
      case  2:    cout << "z.B. einen Text ausgeben";
      break;
      «...»
      default:    Anweisung falls nichts zutrifft
}
```

```
   case 'k':
   case 'K':
     cout << "Die Anweisung reagiert auf Klein- und Großbuchstaben!";
      break;
```

Enumeration (Aufzählungen):

Die werte (wie z.B. Farben) entsprechen Ganzzahlkonstanten.

Diese Konstanten werden als Enumeratoren, die Werte hinter den Konstanten als Indizes der Enumeration bezeichnet.

```
   enum color{
     red,         // Index 0   Falls nichts vorgegeben wird bei 0 angefangen
     green,       // Index 1   und jeweils um eins hochgezählt
     blue,        // Index 2
     yellow = 9, // Index 9   explizite Angabe
     black,       // Index 10  Compiler zählt automatisch um eins hoch
     gray = -1,  // Index -1  negative Zahlen erlaubt
     white,       // Index 0
   };


   int main(){
       color actual_color;
   int input;                                  // Eingabe bis input einen Wert
hat,
       do{                                 // der einem der Indizes entspricht
           std::cin >> input;
       }while(input < red || input > blue);
       actual_color = color(input); // explizite Umwandlung nach color

       switch(actual_color){
           case red:   std::cout << "red"   << std::endl; break;
           case green: std::cout << "green" << std::endl; break;
           case blue:  std::cout << "blue"  << std::endl; break;
       }
```

Der **Wertebereich** einer Enumeration lässt sich aus ihren Indizes ableiten.

Für positive Indizes liegt der Wertebereich zwischen 0 und der kleinsten Zweierpotenz, die größer als der größte Index ist.

Für negativen Indizes muss auch der kleinste Wert größer oder gleich der größten negativen Zweierpotenz sein.

```
   enum A{a, b, c = 15};             //   0 .. 15 ( 0  .. 2⁴ - 1)
enum B{v1 = 1, v2 = 2, v3 = 4}; //   0 .. 7  ( 0  .. 2³ - 1)
enum C{c1 = -10, c2 = 0};        // -16 .. 15 (-2⁴ .. 2⁴ - 1)
enum D{d1 = -10, d2 = 16};       // -32 .. 31 (-2⁵ .. 2⁵ - 1)
```

goto

Ist eine Springanweisung und springt and die durch ein Label gekennzeichnete stelle im code.

Man sollte die Anweisung nicht zu häufig einsetzten, da die Übersichtlichkeit darunter leiden kann. Beim Abbrechen von verschachtelten Schleifen ist sie nützlich.

```
int stop = 0;

for(int a = 0; a < 100; a++)
    for(int b = 0; b < 100; b++)
        for(int c  = 0; c < 100; c++)
        {
            //----
        if(stop)
            goto vortsetzung;
            //----
        }
// Wenn stop ungleich null werdn alle Schleifen unterbrochen und hier vortgesetzt
vortsetzung:   // Label
    // Anweisungen
```

Mehrere Boolsche Werte in einer Variablen:

Mit hilfe der Enumeratoren können meherer boolsche Werte in einer einzigen Variablen gespeicher warden, dazu wird das bitweises-oder-Operator | verwendet.

```
   enum font{
       italic    = 0x01,      // 0b0001 Die Verwendung von Binärzahlen
vermeiden
       bold      = 0x02,      // 0b0010
       underline = 0x04 // 0b0100
   };                                          // Wertebereich 0 .. 7
   int main(){
       font flag = font(0); // Kein flag gesetzt 0000
       char input;

       std::cout << "Kursivdruck? ";
       std::cin >> input;                                                //
0000
       // Binäres Oder zum Setzen eines bla (flag | italic => int)     0001
       if(input == 'j') flag = font(flag | italic);                   //0001
       std::cout << "Fettdruck? ";                       // 0001
       std::cin >> input;                                     // 0010
       if(input == 'j') flag = font(flag | bold);             // 0011
       std::cout << "Unterstreichen? ";                          // 0011
       std::cin >> input;                                        // 0100
       if(input == 'j') flag = font(flag | underline);           // 0111
       // Binäres Und zum Abfragen des bla, dabei ist alles größer Null =
true
       if(flag & italic){                                          // 0001
           std::cout << "Der Text ist kursiv." << std::endl;    // 0111
       }                                                          // 0001
       if(flag & bold){                                                //
0010
           std::cout << "Der Text ist fettgedruckt." << std::endl;     //
0111
       }                                                               //
0010
       if(flag & underline){
           std::cout << "Der Text ist unterstrichen." << std::endl;    //
0100
       }                                                               //
0111
       if(flag == 0){                                                  //
0100
           std::cout << "Der Text wird normal ausgegeben." << std::endl;
```

Die Bitstelle wird mit einem „oder“ | gesetzt und mit einem „und“ & abgefragt. Dabei muss das oder mit dem Bit gesetzt werden also 1,2,4,8,...

Funktionen:

Signatur der Funktion ist die zahlen- und typenmäßige Zusammensetzung der Parameterliste, der Name derFunktion gehört nicht dazu.
Von dieser Signatur hängt ab, ob der Compiler zwei Funktionen als unterschiedlich betrachtet. Da eine Funktion nicht redefiniert werden kann, kann ein gleicher name an zwei Funktionen vergeben werden wenn diese vom Compiler als verschieden betrachtet werden.
Welche Funktion ausgeführt wird entscheidet der Compiler in dem er die Datentypen der Parameter aller Funktionen mit den Argumenten der Aufgerufenen vergleicht, wenn es zur Übereinstimmung kommt wird die Funktion ausgewelt. Typ =T. T und T&, T und T const, T* und T[] sind nicht ausreichend unterschiedlich.

main-Funktion

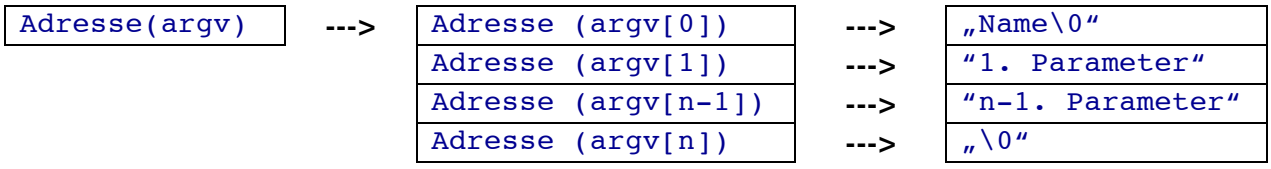
Erlaubt sind:
int main(), void main(), int main(int argc, char *argv[])

Kommandozeilenparameter:
Ein Programm kann wie auch Funktionen mit Anfangsparametern aufgerufen werden, in der Regel werden sie in dem Komandointerprerter beim Programmaufruf angegeben z. B.: sinus 45

- argc (Argument Count) => int argv
- argv (Argument Vector) => char *argv[] = char **argv

Kommandozeilenparameter sind vom Typ Zeichenketten (char).
Ihre Adressen werden vom Compiler in einem Zeigerarray gespeichert. Auf diesen Array zeigt der Zeiger argv. Alos Zeiger auf Zeiger: char **argv

1. Der erste Zeiger verweist auf den Namen des Programms
2. Der Zweite auf bis (n-1) auf die übergebenen Kommandozeilen-Parameter
3. Das letzte Ellement des Zeigerarrays argv[argc] ist der Nullzeiger



Zeiger auf Zeiger Weil die Parameter nicht in einem Block angeordent sind, sondern Beliebig verstreut. Sonst könnte man es in einem Array (Zeiger) unterbringen, wobei jeweils das Nachfolgende Element/Adresse einen neuen Parameter darstellen würde.
Die Parameter werden durch Leer- oder Tabulatorzeichen getrennt eingegeben.

Deklaration:

Vor dem Funktionsaufruf muss die Funktion dem Compiler bekannt sein. Deklaration ist nur eine Bekanntmachung für den Compiler, es wird kein Objekt erzeugt auch kein Speicher Reserviert.

- Lokal: Deklaration innerhalb einer Funktion.
  - Die Funktion ist nur innerhalb dieser Funktion bekann – Gültigkeitsbereich (scope)
- Global: Außerhalb jeder Funktion zu Beginn des Programms

Angabe des Funktionprototyps Typ: Name de Funktion und Parametertype.
Bei einer Deklaration ist es nicht nötig, die Parameternamen mit anzugeben, denn diese sind für den Aufruf der Funktion nicht relevant, wird jedoch empfohlen.

Der Compiler ignoriert die Namen in diesem Fall einfach, weshalb es auch möglich ist den Parametern in der Deklaration und der Definition unterschiedliche Namen zu geben.

Definition:

Darf nicht innerhalb einer anderen Funktion stehen, auch nicht innerhalb main().
Nach dem Compilieren ist das Linken der entstanden Objektdateien zu einem ausführbaren Programm nötig. Der Linker benötigt die Definition der aufzurufenden Funktion.
Eine **Funktionsdefinition umfasst auch die Implementation** der Funktion, d.h. den Code, der beim Aufruf der Funktion ausgeführt werden soll.

Wird die Funktion nach der Definition aufgerufen ist die Deklaration überflüssig.

Funktion kann beliebig oft deklariert, aber nur ein mal Definiert werden.

Besteht ein Programm aus mehreren Quelldateien muss eine Funktion deklariert werden!
Die deklaration kann auch in die Includ-Datei verlagert werden, die Definition wird vom Linker aus der Bibliothek geholt und in das Programm eingebunden. Deshalb werden die Deklarationen in der Header-Datei reingeschrieben und in der main.cpp includiert.
Compilieren: cl oder gcc grog.cpp funk.cpp

Parameter: ist die Angabe des Funktionsprototypen bei der Deklaration oder der Definition.
Argument: konkreten Parameter (Variable oder einen Wert), mit denen eine Funktion aufgerufen wird.

- void f(Parameter){//Anweisungen} // Bei Deklaration und Definition
- f(Argumente/); // Beim Aufruff der Funktion

- Deklaratiion: funktionsname(); mit Parametern => f(int, int);
  - void f(void); kein Rückgabewert, keine Parameter
  - Typ des Rückgabewertes per defaul als „int“ deklariert.
- Definition: Typ f(Typ A, Typ B){...}
- Aufruf: f(A, B);

```
int f (int x); // Der Compiler benötigt die Angabe des Funktionsprototyps == Deklaration

int main(){
int a = f(3);            // Funktionsaufruf mit dem Argument 3,
}                                //der Rückgabewert wird dann in a gespeichert
int f(int x){Anweisung           // Funktionsdefinition
      return Erg;                // nach return wird die Funktion sofort verlassen.
}
```

Rückgabewert

Ein void f() - Funktion hat zwar keinen Rückgabewert, kann jedoch mit der return; – Anweisung beendet werden. Da es **beliebig viele return** – Anweisungen in einer Funktion geben kann kann man diese geschick als kontrollstruckturen einsetzen, ähnlich zu „break“.

Return x; = return (x);

Ist eine Funktion nicht mit void deklariert muss sie eine return-Anweisung haben!

Referenz als Rückgabewert:

Normaler Rückgabewert ist eine temporäre Kopie der lokalen Variablen.
Der Rückgabewert vom Typ: int& f(); ist hingägen ein Alias der Rückgabewariable. Und Damit ein L-Wert und kann auf der linken Seite einer Zuweisung stehen!
Somit wäre der Ausdruck ++f(); / f()=f()+1; möglich!
Nützlich wenn man neben der eigentlichen Ausgabe noch weitere Funktionen implementieren will.
Die Referenzvariable muss immer global, oder static sein!

Benutzen Fremder Funktionen (Bibliotheken):
Implementierung interessiert nicht, deshalb wird die Deklaration und die Definition getrennt.
Der Funktionsprototyp (Format, Parameter) wird in der Deklarationsdatei ***header file*** (Endung: .hpp, .hh, **.h**) angegeben.

Default Werte:

Standardargumente werden in der Deklaration einer Funktion angegeben, da der Compiler sie beim Aufruf der Funktion kennen muss. Auch in der Definition möglich, falls diese sich vor dem Aufruff befindet. Nicht aber in beidern gleichzeitig!!!
Lokale Variablen vom Typ auto dürfen nicht als default-Werte spezifiziert werden.

```
int f(int, int=4, int=0);                    //Deklaration: Die Funktion mit den default Werten versehen

int main(){
   cout << f(1) << endl;                     // 1+4+0 = 5
   cout << f(1, 2) << endl;                  // 1+2+0 = 3
   cout << f(1, 2, 3) << endl;               // 1+2+3 = 6
}          // Es gibt leider keine möglichkeit den 1. und den 3. Parameter zu bestimmen und bei
           dem 2. den Defaultwert gelten lassen.

int f(int a, int b, int c){
   return a+b+c;
}
```

Überladen (***overloading***)

Das Überladen von Funktionen bedeutet, dass verschiedene Funktionen unter dem gleichen Namen angesprochen werden können.
Damit der Compiler die Funktionen richtig zuordnen kann, müssen die Funktionen sich in ihrer Funktionssignatur unterscheiden. Da der Name gleich ist müssen sich hier die Parameter ändern (Anzahl, Typ).

```
int summe(int a, int b, int c, int d) {
  return a + b + c + d;
}
int summe(int a, int b, int c) {
return a + b + c;
```

```
}
int summe(int a, int b) {          //Funkionen unterscheiden sich nur durch die
    return a + b;                  // Anzahl ihrer Parameter
}
```

Inline-Funktion

Um den „Aufruf“ einer Funktion zu beschleunigen, kann in die Funktionsdeklaration das Schlüsselwort `inline` eingefügt werden.

Dies ist eine Empfehlung (keine Anweisung) an den Compiler. Es können kurze Funktionen mit nur wenigen Anweisungen (1-3) ohne Schleifen akzeptiert werden.

Beim Aufruf dieser Funktion keine neue Schicht auf dem Stack anzulegen, sondern den Code direkt auszuführen - den Aufruf sozusagen durch den Funktionsrumpf zu ersetzen.

Die Funktion wird also nicht wie eine normale Funktion aufgerufen, statdessen wird der Aufruf durch die den angepassten Code der Funktion ersetzt.

Der Compiler muss beim Aufruf den Funktionsrumpf kennen, wenn er den Code direkt einfügen soll. Die Funktion muss also bei der Deklaration auch sofort definiert werden, oder man schreibt die Definitionen der inline-Funktionen in eine Header-Datei

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

Übergabe der Argumente:

call-by-value.

Beim Aufruf der Funktion wird der ***Wert* des Arguments** in den Stack des Programms (der Funktion zugeteilten Bereich) **kopiert**.

Ein Werteparameter **verhält sich wie eine lokale** (Gültigkeitsbereich) **Variable**.
Es handelt sich um defacto zwei Variablen, eine ist ein Alias der anderen und kann auch beliebig benannt werden.

Der Kopiervorgang kann bei Klassen (Thema eines späteren Kapitels) einen erheblichen Zeit- und **Speicheraufwand** bedeuten!

Die Änderungen betreffen nur die lokale Kopie und sind für die aufrufende Funktion nicht sichtbar!

Wenn eine Funktion aufgerufen wird: f(iNum) wir der Wert der Variable „iNum“ durch die Definition: void f(int iEingabe) in die Eingabe kopiert. (int Eingabe = iNum)
Zwei verschiedene Variablen iEingang und iNum; Funktion ändert iNum nicht!

```
    void f(int Beliebig);              // Die Angabe des names ist niht obligatorisch

    int main(){
       int a=1;
       f(a);
       cout << iNum << endl;
    }
void f(int b){                 Muss nicht a heißen! zwei verschieden Variablen! b = lokale Variable der funkton f mit
```

```
b++;                        // dem Wert von a initialisiert
      return b;
}
```

call-by-reference:

Im Gegensatz zu call-by-value wird nicht der Wert der Variable kopiert, sondern **Speicheradresse des Arguments übergeben.**

Änderungen der (Referenz-)Variable betreffen zwangsläufig auch die übergebene Variable selbst und bleiben nach dem Funktionsaufruf erhalten. Um call-by-**reference** anzuzeigen, wird der **Operator &** verwendet, wird keine Änderung des Inhalts gewünscht, sollten Sie den Referenzparameter als const deklarieren.

```
   void f(int &a, int &b) {              // Argumente des Aufrufs referenzieren
      Anweisungen...                     // int &a = int& a = int&a = int & a
   }
int main() {
    int x = 5, y = 10;
    f(x, y);                  // Die aufgerufene Funktion ändert die Werte für x und y
    std::cout << "x=" << x << " y=" << y << std::endl;     //neue Werte x/y
   }
```

Alias/**Referenz** nutzen:

Dient dem gleichen Zweck wie Zeiger, bei der Übergabe wird der Wert der Variable nicht kopiert sondern auf die Adresse verwiesen, damit bleibt die Variable auch außerhalb der Funktion nutzbar. Referenzen sind verständlicher als Zeiger, außerdem kann kein Nullzeiger auftreten, was zum Absturz des Systems führt.

```
void f(int&);              // Schon bei der Deklaration der Funktion muss man sagen, dass die Funktion
   int main(){                     // einen Alias verwendet!
       int iNum=1;
       f(iNum);
       cout << iNum << endl;
   }
void f(int& Eingabe){      //Auch wenn man hier als Variable Eingabe verwendet wird mit dem
                           // & nur ein Alias auf die Variable iNum erzeugt, es handelt sich also nur
Eingabe++;                 // um eine Variable
       cout << Eingabe << endl;
   }
```

Wird eine Konstante, oder ein Literal übergeben muss die Funktion so definiert werden, dass sie ein const-Wert erhält. Mit const Übergabe von Referenzen schützt man die Ursprugsvariable, oder den Array vor unerlaubten Veränderungen.

Mit Einer expliziten Konvertierung lässt kann man das umgehen:

```
void f(const int&);
int main(){
    const int konstante=1;
    f(konstante);
    return 0;
   }

void f(const int& konstante){
    int *z = (int*) &konstante;       //explizite Typ-Konwertierung
    *z=3;                  // Kann hierm manipuliert werde, zurück in main
                           wird jedoch der Wert wiederhergestellt
```

```
}
```

```
   f (5);    //An die Function f wird das Literal 5 übergeben
void f(int const &wert)       // Um eine Konstante/ Literal zu empfangen muss
{ Anweisung }                 // die Funktion auch dementsprechend definiert werden.
```

Im Gegensatz zu Variablen werden Arrays immer per Referenz übergeben!

In Verbindung mit **Klassenobjekten** ist die **Übergabe als Referenz auf ein konstantes Objekt** sehr viel schneller.

Für die **Basisdatentypen** ist **die Übergabe als Wert** effizienter.

Call-by-Pointer:

Es gibt jedoch eine Schwachstelle, nämlich wenn ein Nullzeiger übergeben wird.

Außerdem umständlicher als Referenz, dam man derefernzieren muss.

```
void f(int*);      // f(int *a) Schon bei der Deklaration der Funktion muss man sagen, dass die
                   Funktion eine Adresse überliefert bekommt!
   int main(){
            int iNum=1;
            f(&iNum);                 // Hier wird die Adresse von iNum überliefert
            cout << iNum << endl;     // Die Variable iNum wurde inkrementiert ist also jetzt
2
   }
   void f(int* Eingabe){              //Eingabe ist ein Pointer auf die Adresse von iNum
     (*Eingabe)++;                   // * = Objekt und nicht die Adresse um 1 erhöhen
            cout << *Eingabe << endl;
   }
```

Funktionen, die einen Zeiger auf einen konstanten Datentyp erwarten, können auch mit einem Zeiger auf einen nicht-konstanten Datentyp aufgerufen werden.

```
   void function(int const* parameter){}              // konstanter int-Wert
erwartet
int main() {
    int const *zeiger;                                // Zeiger auf nicht-konstanten int
    function(zeiger);                                 // Funktioniert
}
```

Problem: Funktion erwartet: (Zeiger auf Zeiger auf konstanten Datentyp):

Übergeben wird jedoch (Zeiger auf Zeiger auf **nicht** konstanten Datentyp)

```
   // Funktion, die einen Zeiger auf einen Zeiger einen konstanten int erwartet
void function(int const** parameter){}

int main() {
    int** zeiger;       // Zeiger auf Zeiger auf nicht-konstanten int
    function(zeiger);   // Fehler: Typen sind inkompatibel

    // Lösung: explizites casten (Umwandeln) der Konstantheit
    function(const_cast< int const** >(zeiger))
   //gleichbedeutend: const_cast< const*const* > Variable
}
```

Übergabe des Arrays:

Funktionen können **keine Arrays** als Parameter übergeben und auch keine zurückgeben lassen.

Da ein Array eine Adresse hat, wird ein Zeiger übergeben.

Innerhalb der Funktion gibt der Befehl `sizeof(arrayname)` die Größe eines Zeigers in Byte an nicht die Arraygröße in Elementen.

Es ist nicht mehr ein zweidimensionales Array es ist ein eindimensionaler Zeiger.

Eindimensionaler Zeiger = [Array]

```
void funktion(int *array);  // (Formal-)Parameter für Array
void funktion(int array[]); //Auch möglich da Array = const-Zeiger
void funktion(int array[5]);      // Größenangaben werden ignoriert
```

Eine bestimmte Stelle kann man jedoch somit übergeben:

```
void funktion(int &array[&3]);
void funktion(int (array+2));
```

Mehrdimensionaler Zeiger [Array]

Ab der zweiten Dimension geben Sie also tatsächlich Arrays an, somit müssen Sie natürlich auch die Anzahl der Elemente zwingend angeben. Daher können Sie sizeof() in der Funktion verwenden, um die Größe zu ermitteln.

Typ (*)[E2][E3][E4]... // Zeiger auf das erste Arrayelement

```
void funktion(int (*array)[8]); //Zeiger auf ein Array, Beachte(*array)
void funktion(int array[][8]);
void funktion(int array[4][8]);

   // Zugriff
   (*array)[k]   =    array[i][k];    =    *(*(array+i)+k);

   // Aufruf
   funktion(array);
```

Beachten! Es wird ein Zeiger und kein Array übergeben!

Zeiger ist ein modifizierbarer L-Wert, Array ist jedoch nicht modifizierbar, die Adresse kann nicht geändert werden!

```
void f(short array[], int n);    // auch *array zulässig

int main(){
    int n;
    short array[]={11,22,33,44,55,66,77};
    n=sizeof(array)/sizeof(array[0]); // im main ist es ein array,
                          nur hier! kann sene Größe bestimmt werden.
    f(array, n);       // Aufruf auch mit f(*array / &array[0]); möglich. n = 7
return 0;
   }
//********** f-Definition ************
```

```
void f(short *array,int n){
      cout << sizeof(short) << endl;       // Shortgröße = 2 Byte
      cout << sizeof(short *) << endl;     // Zeigergröße = 8 Byte

// Achtung: da es sich NICHT um einen Array handelt, kann auch seine
Größe nicht bestimmt werden. Folgende Anweiseung gibt die
(Zeigergröse)/(Short-Größe) aus => 8/2 = 4 und nicht 7!!!
    cout << sizeof(array)/sizeof(array[0]) << endl; // = 4!

for (int i=0; i < sizeof(array)/ sizeof(array[0]); i++) { //FEHLER!
    for (int i=0; i < n; i++) {
    cout << *array++ <<" "; // Da ein Zeiger und kein Array kann er
           hier inkrementiert und sich selbst zugewiesen werden.
    }
}
```

C++-Arrays:

Seit C++11 gibt es den Header <array> in welchem eine gleichnamige Datenstruktur definiert ist.

Die C++ - Arrays werden verwendet wie gewöhnliche C-Arrays, haben aber eine andere Deklaration. Ein Array mit 8 Elemente vom Typ int wird wie folgt definiert:

```
   #include <array>
// ...
std::array< int, 8 > variablenname;
```

Für **mehrdimensionale** Arrays kann man statt int als Datentyp wieder ein Array angeben.
Die Deklaration ist also etwas aufwendiger als bei C-Arrays, dafür kann ein solches C++-Array aber ganz normal an Funktionen übergeben werden.

```
   void funktion(std::array< int, 8 > parameter); //Als Kopie übergeben
void funktion(std::array< int, 8 > const& parameter); // Als Referenz (const)
   //Übergabe eines mehrdimensionalen Arrays
void funktion(std::array< std::array< int, 8 >, 4 > const& parameter);
```

C++-Array-Objekte haben eine Funktion names `size()`, welche die Anzahl der Elemente zurückgibt.

Funktionen mit unbestimmter Parameterzahl

```
Definition/Deklaration:
Typ_Rückgabewert Funktionsname(Feste_Parameter , ...)
```

```
// Argumentenzeiger (Argument-Pointer) definieren
va_list ap;
```

```
// Argumentenzeiger mir der Adresse des ersten optionalen Parameters initialisieren
va_start(Argumentzeiger, letzte_feste_Parameter);
```

1. Stellt den Wert des optionalen Parameters zur Verfügung. Ist der Parameter ein Zeiger wird eine Adresse zurückgegeben.
2. Modifiziert den Argumentenzeiger so, dass er auf den zweiten optionalen Parameter zeigt
va_arg(Argumentenzeiger, Typ_optionaler_Parameter);

Der Stack wird aufgeräumt und die Adresse des Argumentenzeigers auf NULL gesetzt
va_end(Argumentenzeiger);

```
//#include <string.h>           // strcat
//#include <stdarg.h>           // va_start, va_end, ...

char* scat(int anzahl, char *s1, ...);
int main(){
    char str1[128] ="erster String: S1 ";   // unbedingt beachten,
dass der erste String groß genug ist
    char str2[] ="optionaler Parameter";

    cout << str1 << endl;

    scat(3, str1, "angefuegt ", str2);
    cout << str1 << endl;
    return 0;
}
char* scat(int anzahl, char *s1, ...){  // Analog zu strcat nur
werden beliebig viele Strings zusammengefügt. s1 = String Nr.1

    int i;
    va_list ap;
    va_start(ap, s1);          // Zeiger auf Argumente definieren

    for(i = 0; i < anzahl; i++)
      strcat(s1, (va_arg(ap, char*)));    // Initialisierung des
Argumentenzeiger mit der Adresse des ersten optionalen Parameters

    va_end(ap);
    return (s1);
}
```

Funktionszeiger (Zeiger auf Funktionen):

Zeiger können nicht nur auf Variablen, sondern auch auf Funktionen verweisen.

F = &F = der Name der Funktion ist eind constanter Zeiger auf den Anfang des Funktionscodes im Code-Segment.

Keinerlei Zeigerarithmetik gestattet!

Deklaration: statt des Funktionsnamens wird der Variablennamen mit Stern in Klammern angegeben.

*int f(double d); => int (*fz)(double d);*

// da die Priorität: **()** > * müssen die klammern angegeben werden, sonst heißt:

int *fz(double) eine Funktion mit einem Rückgabewert vom Typ „int *“.

```
int multiplication(int a, int b){
    return a*b;
}
```

```
int division(int a, int b){
    return a/b;
}
int main(){
    int (*rechenoperation)(int, int) = 0; // Zeiger auf Funktion mit 0 initialisiert
    rechenoperation = &multiplication;          // Adresse der Funktion
       Ergebnis = (*rechenoperation)(40, 8)  // Ergebnis = rechenoperation(40, 8)
                                              // geht auch aber nicht empfohlen
    std::cout << Ergebnis << std::endl;
   // Das Objekt (*)-Zeiger ausgegeben. Da es sich um einen Funktionszeiger handelt werden die
    // Parameter mit übergeben und anstatt der Funktion, wird der Returnwert ausgegeben.
       rechenoperation = &division;      // Andere Funktion wird dem Zeiger zugewiesen
    std::cout << (*rechenoperation)(40, 8) << std::endl;
}
```

Der Adressoperator ist nicht zwingend zum Ermitteln der Funktionsadresse notwendig. Gleiches gilt bei der Dereferenzierung: ein expliziter Stern vor dem Funktionszeiger macht deutlich, dass es sich um eine Zeigervariable handelt.

```
double plus_(double a, double b);     // plus ist vergeben => plus_
double minus_(double a, double b);
double mal(double a, double b);
double geteilt(double a, double b);

int main(){
    double  a, b;
    int auswahl=0;

// int(*zf)(double a, double b) = f;

    double (*z[4])(double, double)={plus_,minus_,mal,geteilt};
// cout << "a eingeben" << endl;
    cin >> a;
//    cout <<"b eingeben" << endl;
    cin >> b;

    do{
    // cout << "oeration wählen" << endl;
    cin >> auswahl;
    }while(auswahl>3 || auswahl<0);
    cout << z[auswahl](a,b);
return 0;
}
// Funktionen Definieren
double plus_(double a, double b){
    return (a+b);}          // da Funktion als double Deklariert ist muss es einen
                            Rückgabewert geben!!
double minus_(double a, double b){
    return (a-b);}
double mal(double a, double b){
    return (a*b);}
double geteilt(double a, double b){
    return (a/b);}
```

Rückgabe von mehreren Variablen:

Verwendung einer Klasse als Typ für den Rückgabewert. Ein Objekt der Klasse kann viele Datenelemente enthalten.

Da Funktionen nur einen Rückgabewert haben hilft ein Trick um mehrere Variablen auszugeben:

```
void f(int, int&, int&);

int main(){
   int Eingabe = 1;
   int Ausgabe1;
   int Ausgabe2;

   f(Eingabe, Ausgabe1, Ausgabe2);          //Ausgaben werden als Platzhalter an die Funktion überliefert

   cout << Eingabe << "\t" << Ausgabe1 << "\t" << Ausgabe2 << endl;
}
void f(int Eingabe, int& refAusgabe1, int& refAusgabe2){      //Die Platzhalter referenzieren zurück
   refAusgabe1 = Eingabe +1;                                  auf die Variablen, da wo die Funktion
   refAusgabe2 = Eingabe +2;                                  aufgerufen wurde
}
```

Die Arrays werden der Funktion als Zeiger (*) übergeben.
Funktion erhält als Parameter die Startadresse des Arrays (ein Zeiger) und seine Größe.
Wenn das Array nur gelesen werden soll, deklariert man die Werte, auf die der Zeiger zeigt, als `const`.

```
   void f(int const* array, int const arrayGroesse) {
.     Anweisungen
}

int main() {
      int array[] = { 3, 13, 113 };
      int n = sizeof(array) / sizeof(array[0]);

      f(array, n);
}
```

Funktion Overloading:

Ist wenn es zwei oder mehr funktionen mit dem gleichen Namen gibt.
Sollten sich die Funktionen nur durch den Argumententyp unterscheiden helfen
Function Templates:

```
    int f(int, int);
    double f(double, double);
template <typename DTParam> DTParam f (DTParam , DTParam );      //Funktion mit eigenem
                                                                  Datentyp deklarieren
    int main(){
        int iA=2, iB=3;                          //An dem Aufruf der Funktionen ändert sich nichts
        f(iA, iB);
        double dA=2.2, dB=3.3;
        f(dA, dB);

        cout << f(iA, iB)<< "\n"
             << f(dA, dB) << endl;
    }
    //Zwei gleiche Funktionen, die sich nur aufgrund des Datentyps unterscheiden
    int f(int iA, int iB){
        return iA + iB;
    }
    double f(double dA, double dB){
        return dA + dB;
    }
    //Besser Template erzeugen!
    template <typename DTParam> DTParam f(DTParam A, DTParam B){ //
        return A + B;
    }
```

Sollte bei einem konkreten Datentyp eine andere Funktion aufgerufen warden, kann diese spezialisiert warden:

Callback-Funktion:

Eine **Callback-Funktion** ist eine Funktion, die einer anderen Funktion als Parameter übergeben und von dieser unter gewissen Bedingungen aufgerufen wird.
Eine Callback-Funktion kann auch in einem Objekt gespeichert und von diesem unter gewissen Umständen aufgerufen werden.

Funktions-Aufrufe können unabhängig von der eigentlichen Funktion implementiert werden. Die eigentliche Funktion kann in anderen Modulen implementiert werden. Die Callback-Funktionen werden also nicht statisch eingebunden, sondern erst zur Laufzeit des Programmes.

Typische Beispiele für Callback-Funktionen sind sog. Event-Handler:
Ein Objekt, z.B. ein Button, stellt ein Property(Eigenschaft) *OnClick* zur Verfügung, welchem eine Callback-Funktion zugewiesen werden kann.
Sobald der User auf diesen Button klickt, wird die zugewiesene Callback-Funktion vom Button aufgerufen. In dieser Funktion ist die entsprechende Aktion implementiert.
Durch diesen Mechanismus muss der Button nicht wissen, was für eine Aktion beim Klicken ausgeführt werden muss. Die Implementation des Buttons ist somit von der Verwendung getrennt. Die selbe Button-Klasse kann von vielen verschiedenen Modulen

verwendet werden, wobei jedes Modul eine andere Aktion im zugehörigen Button als Callback-Funktion installieren kann.

Diese Art der Programmierung hat den Vorteil, dass das Programm nicht in einer Schleife auf bestimmte Ereignisse warten muss (z.B. auf einen Mausklick). Statt dessen werden bei den Ereignissen installierte Callback-Funktionen gerufen.

Rekursive Funktion

Funktion die sich selbst aufruft. Das hat zum Nachteil, dass mit jedem Aufruf (Rekursion) die Funktion auf dem Stack gestappelt wird und da nach dem LIFO-Prinzip die Programme nicht entfernt werden, sondern viel Speichplatz belegen kann es zum stack-owerflow kommen.
Außerdem sind die Rekursionsaufrufe langsammer als die iterativen Aufruffe der Schleifen.

Die Funktion besteht aus zwei Teilen einen rekursiven, den Teil bevor die Funktion sich selbst wieder aufruf und dem Rückweg. Das ist der Teil nach dem Aufruf der Funktion zu dem das Programm nach dem Wendepunkt zurückkommt.

```
void zaehler(int n);
int main(){
    zaehler(3);
    return 0;
}

void zaehler(int n){
Rekursiver Teil
    cout << n << " ";      // wird hochgezaählt, da in der Rekursion
    if (n > 0) {
        zaehler(n-1);
    }
Rückweg
    cout << n << " ";      // Wird runtergezählt, da auf dem Rückweg
}
```

Häufige (eine der wenigen sinnvollen) Anwendung in Baumstruckturen.

Klassen

Anstatt einfach Daten (Variablen) zu haben, die von Funktionen manipuliert werden, fasst man die Daten und die darauf arbeitenden Funktionen zu einem neuen Konstrukt zusammen. Dieses Konstrukt nennt sich „Klasse“ und das Zusammenfassen wird als Kapselung bezeichnet.

Unterschied zwischen struct und class darin, dass bei einer mit **struct** definierten Klasse (struct=class) alle Daten **public** und beim Schlüsselwort **class** alle Daten **private**.

- Eine **Klasse** ist ein benutzerdefinierter(vortgeschrittener) Datentyp.
- Eine Variable vom Typ einer Klasse beinhaltet alle in der Klasse deklarierten Variablen. Somit verbraucht eine Klassenvariable theoretisch so viel Speicherplatz, wie die Variablen, die in ihr deklariert wurden. Für die genaue Größe den `sizeof`-Operator auf die Klasse oder eine Variable vom Typ der Klasse anwenden.
- Eine Klasse hat üblicherweise einen Namen und besteht aus Variablen und Funktionen, welche als **Klassenmember** bezeichnet werden.
  Die Notation ist im Allgemeinen so: var_, m_var, mVar, ...
- Um etwas besser zwischen einer Variable innerhalb einer Klasse und einer Variable vom Typ der Klasse unterschieden zu können, werden Variablen vom Typ der Klasse als **Objekte** der Klasse bezeichnet.
- **Datenobjekte** sind Variablen die innerhalb einer Klasse difeniert wurden.
- Ein Funktionsmember bezeichnet man auch als **Methode** der Klasse.
- Als **Schnittstelle** bezeichnet man die Member der Klasse, die von außen sichtbar sind. Dies werden meistens eine Reihe von Methoden sein, während Variablen nur sehr selten direkt sichtbar sein sollten.
- Klassen haben zwei spezielle Methoden, die beim Erstellen (**Konstruktor**) bzw. Zerstören (**Destruktor)** eines Objektes vom Typ der Klasse aufgerufen werden. Der Name des Konstruktors ist immer gleich dem Klassennamen, der Destruktor entspricht ebenfalls dem Klassennamen, jedoch mit einer führenden Tilde (~).
- Innerhalb einer Klasse kann es verschiedene Sichtbarkeitsbereiche geben, die darüber entscheiden, ob ein Member nur innerhalb (`privat:`) von Methoden der Klasse oder auch von außerhalb (`public:`) aufgerufen werden kann (Schnittstelle).
- Typischerweise werden Variablen `private` deklariert, während Methoden `public` sind. Eine Ausnahme bilden Hilfsmethoden, die gewöhnlich im `private`-Bereich deklariert sind, wie etwa die `init()`-Methode, die von verschiedenen Konstruktoren aufgerufen wird, um Codeverdopplung zu vermeiden.
- Der einzigste Unterschied zwischen Strukturen und Klassen ist in C++, dass Klassen implizit `private`, während Strukturen implizit `public` sind.

  - **Datenabstraktion:** Daten und Funktionen gehören zu einer Klasse (Datentypen)
  - **Kapselung :** Elemente eines Objekts einer Klasse sind vor dem falschen Zugriff geschützt, zugriff nur über Schnittstellen. (private)
    Man muss nicht wissen wie eine Klasse (Typ) aufgebaut ist man muss nur die Schnittstellen kennen.
  - **Vererbung:** Das neue Objekt erbt die Eigenschaften (Daten) und Methoden (Funktionen) des vorhandenen Objekts.
  - **Polymorphie:** Variation durch Vererbung und Modifikation der Klasse

- Eine Klasse Basisklasse „Figur“, Methode „volumen()“: dreicek::volumen, kreis::volumen „..

Instnzieren = Erstellen eines Objektes einer Klasse

Deklaration

```
class Mensch {        //Klassendefinition
// Eigenschaften (Datenelemente) der Klasse Mensch
// Definition des Datentyps (Klasse) und nicht des Elements!
    char name[30];
    unsigned int alter;
    unsigned int alter = 0;      //Fehler!: Initialiesierung hier nicht möglich
    bool geschlecht;             //0 = männlich; 1 = weiblich

// Fähigkeiten (Methoden) der Klasse Mensch
void sehen( const char* objekt );
void hoeren( const char* geraeusch );
void riechen( const char* geruch );
// Einen Menschen mit allen Daten erzeugen
void erzeuge( const char* n, unsigned int a, bool g );
};   // Impliziter Destruktor der Klasse Mensch, inhalt nur bis hier her bekannt.
```

Keinen Speicher reserviert – es handelt sich lediglich um eine Anweisung für den Rechner, was die Klasse »Mensch« alles darstellt. Der Rechner weiß hierbei, wie viel Speicherplatz er für ein Objekt »Mensch« reservieren muss, `sizeof(Klassenname);`.

Da der Speicher nicht reserviert wird ist auch die Inittialisierung nicht möglich!

const-Datenobjekte müssen bei der ihrer Definition initialisiert werden. Bei Klassen wird bei der Definition noch kein Speicher allokiert, deswegen können const-variablen auch nicht initialisiert werden. Dazu benötigt man einen Konstuktor.

Man kann auch eine leere Klasse erzeugen um einen Zeiger auf diese Klasse anzulegen. Definition erfolgt später.

```
class Mensch;          // Ohne {}; !
```

Methode definieren

Definition innerhalb der Klasse. Die Funktion ist implizit eine Inlinefunktion. Besser für kleine Funktionen. Die Funktion muss jedoch vorher bekannt sein, Header-Datei.

```
class Mensch {
// Eigenschaften (Datenelemente) der Klasse Mensch

// Fähigkeiten (Methoden) der Klasse Mensch

// Definition der Elementfunktion sehen in der Klasse
    void sehen( const char* objekt ) {
    // Anweisungen für die Funktion
    }
};
```

In der Praxis wird gewöhnlich, wegen der Übersichtlichkeit, die **Definition** einer Elementfunktion **außerhalb der Klasse** vorgenommen. Hierbei muss man jedoch auf den **Gültigkeitsbereich der Methode angeben**, da die Funktion ein lokales Objekt ist - Scope-Operator (Bereichsoperator) :: .

```
class Mensch {
// Eigenschaften (Datenelemente) und Fähigkeiten (Methoden)
};
// Definition außerhalb der Klasse
Typ Klassenname::funktionsname( parameter ) {} // qualifizierter Name ::

void Mensch::sehen( const char* objekt ) {
        // Anweisungen
}
```

Eine Methode hat immer Zugriff auf sämtliche Elemente der Klasse. Die Objekte der selben Klasse müssen nicht explizit übergeben werden.

Objekte deklarieren

```
Mensch frau, mann, gruppe[10];  // Man kann auch mehrere Objekte
deklarieren
```

Mit einer solchen Deklaration wird jetzt Speicher für die Objekte »frau« oder »mann« reserviert.
Datenelemente (für jedes Objekt): „name“, „alter“ und „geschlecht“.
Code der Klassenmethoden nur einmal im Speicher abgelegt

Inhalt der Daten zunächst undefiniert. Wenn das Objekt als static, oder global definiert wird, wird der Inhalt standardmäßig mit 0 belegt.

Objekte kann man auch dierekt bei der Klassen definition erzeugen:

```
class Mensch {
// Klassenmember
} a, b, c[10];        //Objekte direkt bei der definition erzeugen
```

Lokale Klasse

Klasse die inerhalb einer Funktion difeniert ist

```
void f(){
    int a =1;
    static int b=2;

//Für die Klasse die inerhalb eider Funktion definiert ist sind nur die static variablen bekannt.
class A{                          // Lokale Klasse
          void show(){
               cout << a;         // Fehler, da a unbekannt
               cout << b;         // Da b static ist es bekannt
          }
    };
}
```

Zugriffspezifizierer

Abschotten der Klassenelemente vor dem allgemeinen Zugriff wird auch als Kapselung bezeichnet.
Zugriffsbeschränkungen der spezifizierer:

private:	Nur Methoden der Klasse und *friends*
protected:	Methoden der Klasse, *friends* und abgeleitete Klassen
public:	Keinerlei Zugriffsbeschränkungen für das Klassenelement

Objekte erzeugen und initialisieren

Während der Klassendeklaration wird kein Speicherplatz reserwiert, folglich dürfen dort die Datenelemente auch nicht initiiert werden. Wie dann?

Parameterliste

Datenelemente mithilfe der Parameterliste initiieren eignet sich **nur** für Objekte deren Datenelemente ***piblic*** sind.

```
class Zeichenkette{ public: int laenge; char* zeichen;
};
    Zeichenkette a ={3, "ABC"};       //Initialisierung des Objektes
    Zeichenkette b[3] ={ 4, "Eins",
                         4, "Zwei",
                         0, '\0'};  //Defaultwerte Nullwerte des Typs
```

Zweite Möglichkeit ist es das Objekt mit den Werten eines anderen Objektes zu initialisieren.

```
Zeichenkette c  = a;
Zeichenkette c(a);          // geht auch!
Zeichenkette d[] = b;       // Geht nicht!
```

Aufpassen!: Die Werte für das char-Array „zeichen“ werden nicht kopiert. Es wird nur die Adresse übertragen! Somit haben zwei verschiedene Objekte ein untrennbarees Datenobjekt!
Ändert man „zeichen“ in a, wird c mitverändert und andersrum. => **Flache Kopie**

Um das zu umgehen muss man den Speicherbereich duplizieren und der neuen Adresse (b-Objekt) den Wert des char-Strings „zeichen“des a-Objektes zuweisen (siehe Kopierkonstruktor).

Konstructor:

- Ein Konstruktor ist eine Methode (Elementfunktion) welches ein Objekt erzeugt, er wird immer dan Aaufgerufen, wenn ein Objekt erzeugt wird. Der Name der Methode = Name der Klasse.
- Primäre Aufgabe des Konstruktors ist die Initialisierung eines Objektes, seine Aufgabe ist Speicherplatz für die Attribute bereitzustellen. Auch wenn eine Objekt nicht initialisiert wird wird er jedoch mit definierten Anfangswerten versehen (Default-Konstuktor)
- Konstruktoren werden oft als Inline-Funktionen implementiert (direkt in die Klassendeklaration geschrieben).

- Attribut = Merkmal/ Eigenschaft eines Objekts:
  Objekte (Fenster, Buttons, Laufleisten, Menüs, …) besitzen verschiedene Eigenschaften (Farbe, Größe, Ausrichtung, …). Diese Eigenschaften eines Objekts heißen **Attribute**.
- Ein konstruktor wird beim Erzeugen eines Objektes IMMER aufgerufen, auch wenn implizit:
  Dabei handelt es sich um einen: Default-Standard- oder Default-Kopierkonstruktor, oder um einen Explizit für diese Klasse vereinbarten Konstruktor.
- Wird das **Objekt** mit new **dynamisch** auf dem Stack erzeugt **muss es explizit aufgerufen werden**.
- Konstruktor muss *public* deklariert sein!

```
class Auto{
    int ankgroesse;    //Datenelemente sind hier per default private
    int tankinhalt;    // Keine Initialisierung hier!
    int verbrauch;     // Verbrauch(7); geht nicht!
    int a;
```

// Da hier noch kein Speicher reserviert wird können die Datenelemente auch nicht initialiwiert werden! Damit kann auf die Elemente auch nicht zugegriffen werden.

```
public:

//******* Konstrucktor-Deklaration **********
Auto(int tankgroesse, float tankinhalt, float verbrauch, int x);
};

// ******* Konstrucktor-Definition außerhalb der Klasse ******
Auto::Auto(int tankgroesse, float tankinhalt, float verbrauch){

// This nötig damit die Parameter nicht die Datenelemente überdecken
    this->tankgroesse = ankgroesse;
    this->tankinhalt = tankinhalt;
    this->verbrauch = verbrauch;
           a = x;

//Andere möglichkeit wäre:
Auto::tankgroesse = tankgroesse;        //analog zum :: Operator um auf
}                                       globale Variablen zuzugreifen.
```

Initialisierungsliste:

Sollen Konstanten, Referenzen, oder Klassenobjekte initialisiert werden muss es über die so genannte Initalisierungsliste erfolgen. (Oder im Rumpf des Konstruktors)

- Eine Konstante kann kann nur initialisiert werden.
- An eine Referenz kann nur zugewiesen werden, wenn sie zuvor initialisiert wurde.

- Die Liste wir vor dem Konstruktor-Rumpf mit einem **:** eingeleitet.
- Die Initialisierungswerte werden in runden Klammern geschrieben ();

```
class bla{
int x;
const int c;      // Inerhalb einer Klasse erlaubt, muss aber durch den Konstruktor
                  initialisiert werden!
int& r;           // Gleiches gilt für eine Konstante

bla(): r(x),c (1){ // Nur so geht es!!!
        //c = 1;      // Fehler!
        //r = a;      // Fehler!
    }
};
```

Desweiteren ist die Initialisierungsliste dann erforderlich, wenn Datenelemtene initialisiert werden sollen, wobei das Datenelement wiederum ein Objekt einer Anderen klasse ist.

Wenn ein Datenelement der Klasse B ein Objekt der Klasse A ist, kann man es nicht einfach initialisieren, da ein Objekt ja von dem Konstruktor erst erstellt werden muss.
Das heiß, dass bei der Initiierung des Datenelemnts der Klasse B der Konstruktor der Klasse A aufgerufen wird, den man wiederum Parameter übergeben kann und so initieren kann.
Dieser Aufruf muss in der Initiierungsliste erfolgen.

```
class A{
    int a, b;
public:
    A(int aob_1, int aob_2){a=0; b=0} //Standardkonstruktor
    A(int aob_1, int aob_2){a=aob_1; b=aob_2} //Konstruktor A_2
                    Der Kons. A wird mit einem Parameter aufgerufen
};
class B{
    int x;
    A aob;          // für die Erzeugung des Datenelemnt-Objektes
                    ist der Konstruktor A zuständig
public:
    B(): x(3), aob (7,13)  [oder aob()] {} // Initialisierungsliste:
Initialisierung des Datenelementes aob, das Wiederum ein Objekt der
Klasse A ist.

Der Konst. A wird mit dem Parameter 7 und 13 aufgerufen diese er den
Variablen a und b zuweist.

Wenn keine Parameter übergeben werden, muss aob() nicht explizit
aufgerufen werden. Der Standardkonstruktor wird auch so aufgerufen.
};
```

```
          // class Name_der_Klasse(Datentyp)
class Bruch{
   int zaehler_;
   int nenner_;

   public:

   //**** Initialisierungsliste ****
   (Andere möglichkiet die Datentypen zu initiiren)

// Innerhalb der Klassendeklaration
Bruch(int z, int n):          // Konstructor, Klassenprototyp (Deklaration)
      zaehler_(z),            // Initialisierung von Membervariablen:    zaehler_
      nenner_(n)                                                        nenner_
      Bruch *BZeiger_;        //Auch Zeiger auf die Klasse sind möglich
      {leer}                  // Rumpf, Initialisierung erfolgt davor

   // Initialisierung einer Variable innerhalb der Klasse nur var(x); zulässig, ~~nicht var =x;~~


private:

};
   //*************** Konstruktor Aufruf ***************
int main(){
      Bruch x(7, 10);              // Der Konstrucktor wird implizit aufgeruffen
      ~~Bruch x();~~               // Nicht erlaubt (siehe unten)
```

```
      Bruch y = Bruch(3, 6);       // Expliziter Konstruktionsaufruf für
                                   das Objekt y der Klasse Bruch
// Der Konstruktor für ein dynamisch erzeugtes Objekt muss explizit
aufgerufen werden. Beim Fehler in der Allokation wird kein Objekt
erzeugt, an z wird ein Null-Zeiger übergeben.
      Bruch *z = Bruch(3, 6);

}
```

Definition mit default-Parametern

Enthält die Konstruktordefinition Paarameter, so muss auch deren genaue Anzahl beim Konstruktoraufruf vorhanden sein!

Um jedoch einen Konstruktor mit weniger, oder gar keinen Parametern aufzurufen muss man den Parametern default-Werte vergäben.

```
   class Bruch{
public:
    Bruch(int z=0, int n=1);       // Konstruktor-Deklaration mit default Parametern
private:
    int zaehler_;                  // Deklaration
    int nenner_;
};
   // Definition auserhalb der Klasse

Bruch::Bruch(int z, int n){
      zaehler_ = z;                          // Zuweisung von zaehler_
      nenner_ = n;                           // Zuweisung von nenner_
}
// ODER
   Bruch::Bruch(int z, int n):     // Definition, Klasse::Konstructor
      zaehler_(z),                           // Initialisierung von zaehler_
      nenner_(n)                             // Initialisierung von nenner_
      {}

int main(){
// Es ist eine Objektdefinition (Initalisierung), kein Funktionsaufruf!
//Implizit:
Bruch a(7, 10);  // a = 7/10
      Bruch b(7);// b = 7          auch Bruch b = 7; erlaubt
      Bruch c;          // c = 0
      Bruch c();        // Forsicht! Es ist keine Objektdefinition
                        sondern eine Funktionsdeklaration!
// Explizit:
Bruch objekt1 = Bruch(7, 10);
Bruch objekt2 = Bruch(7);
Bruch objekt3 = Bruch();     // So erlaubt

}
```

- Initialisieren nur mit *var(x);* , Zuweisung auch mit *var = x;* möglich
- **Initialisierung** erfolgt innerhalb der Klasse **vor dem Konstruktorrumpf**.

- Initialisierung ist für die Basisdatentypen gleichschnell wie die Zuweisung, bei komplexen Datentypen ist die Initialisierung jedoch schneller.

Arrays können nicht initialisiert werden, sie müssen immer im Konstruktorrumpf mittels Zuweisung ihren Anfangswert erhalten.

Wird es denoch benötigt muss man auf die Klasse `std::vector` (Array ähnlich) aus der Standardheaderdatei `vector` zurückgreifen.

Um Performanceeinbusen zu vermeiden das `std::vector`-Objekt zunächst mittels der Methode `reserve(n)` Anweisen für `n` Elemente Speicher zu allozieren und anschließend Ihre Elemente mittels der Methode `push_back(element)` hinzufügen.

Überladen des Konstruktors

Der Konstruktor kann wie eine gewöhnliche Funktion **überladen** werden.

Es ist jedoch besser die gleichen Teile mit `init()` zu schreiben:

```
   class A{
public:
    A(double x);
    A(int x);
private:
    void init();          // Deklaration
    int x_;
    int y_;
};
A::A(double x):           // Konstruktor 1 für double mit Initialisierung
    x_(x) {
    init();               // Für den Wert y_ wirt die Initialisierungsfunktion init() aufgerufen.
    }
A::A(int x):              // Konstruktor 2 für int mit Initialisierung
    x_(x) {
    init();               // Für den Wert y_ wirt die Initialisierungsfunktion init() aufgerufen.
    }
void A::init(){
    y_=7000;       // Zuweisung: Ändert man der Wert für y_ hier wird es überall geändert!
}
```

Standardkonstruktor:

Konstruktor der keine Parameter erwartet. Auch wenn Parameter angegeben werden können – default-Werte. Sinnvoll ist nur ein Standard-Konstruktor pro Klasse.

Initialisierung nur statischer und globaler Objekte, jedoch keiner non-static Objekte, deren Werte undefiniert sind.

```
   class Bruch{
public:
    Bruch();                          // Deklaration Standardkonstruktor
    // Bruch(int z = 0, int n = 1);      // Ist auch ein Standard-Konstruktor
   // Wenn man einen default-Standard-Konstruktor definiert, so wird der
Standartkonstruktor des Compilers nicht aufgerufen, das heißt eine Angabe ohne Parameter ist
nicht möglich!
private:
  int zaehler_;
  int nenner_;
};
```

```
   Bruch a;                 // globales Objekt      => zaheler_= 0, nenner = 0

int main(){
static Bruch a;             // static- Objekt       => zaheler_= 0, nenner = 0
       Bruch a;             // lokales Objekt       => zaheler_= ???, nenner = ???
}
```

Kopierkonstruktor:

Erstellt ein Objekt und initialisiert es mit den Werten eines bereits vorhanden Objektes derselben Klasse. Standartkopierkonsturktor kann nur elementenweise kopieren kann keine default-Parameter besitzen.

```
   K::K(K const& vorhandenes_Objekt);          // wikibooks
```

```
   K::K(const K& vorhandenes_Objekt);          // Buch
```

Standart-Kopierkonstruktor kann nur elementweise kopieren, das heißt, dass beim Kopieren von Arrays, oder Zeigern die nur Adresse derselben übertragen werden und nicht deren Werte! Deshalb definiert man für diese Aufgaben einen eigenen Kopierkonstruktor.

```
class strg{       // Klasse String
    int len;          // Länge des Strings
    char *string;     // Zeichekette = char [];

public:
    strg(){           // Standardkonstruktor
        len =7;
        string = "Default"; }
    strg(const strg& s);          // Deklaration Kopierkonstruktor
    void show(){cout << string << ": " << &string << endl;}
};

strg::strg(const strg& s)         // Definition Kopierkonstraktor
{
    const int ERROR = 1;
    len = s.len;
    string = new char [len+1];          // Speicherplatz für den String
                                        allokieren (+1 wegen '\0')
     if(string)                   // Allokieren OK, kein Nullzeiger
      strcpy(string, s.string);
     else
     {
      cout << "Fehler bei der Speicherplatzbelegung";
      exit(ERROR);                // Abbruch des Programms
     }
}

int main(){

    strg a;             // Objekt a erzeugen
    strg b=a;           // Objekt b erzeugen und mit den Werten des
                        Objektes a initiieren
// Es existieren nun zwei verschiedene Datenelemente „String“ die
mit dem gleichen Inhalt belegt sind
```

```
    a.show();          // Einhalt: Default, Adresse: 0x7fff5fbff8d0
    b.show();          // Einhalt: Default, Adresse: 0x7fff5fbff8c0


return 0;
}
```

Kopierkonstruktor wird auch aufgerufen, wenn ein Objekt einer Funktion als Parameter übergeben wird, oder von einer Funktion als Rückgabewert zurückgeliefert wird.

Man kann ein Objekt auch per Referenz übergeben. void f(K& a) {}

Konvertierungskonstruktor

Konvertierungskonstruktor (conversion constructor) ist ein Konstruktor einer Klasse K der Objekte anderer Datentypen in den Datentyp der Klasse K umwandelt.

Der Anwendungsbereich ist eingeschränkt, da nur die Umwandlungen ine eine Klasse, jedoch nicht in Datentypen möglich sind.

Konvertierung eines Klassenobjektes in ein Datenobjekt sind nicht möglich!

Merkmale des Konvertierungskonstruktors:

- Mit einem einzelnen Parameter aufrufbar. Alle anderen Parameter müssen über default-Werte verfügen.
- Der erste Parameter ist nicht vom Typ der Klasse.

Konvertierungskonstruktor wird stets dann (implizit) aufgerufen, wenn der benötigte Datentyp nicht zum Typ der Klasse passt. Aufruf einer Funktion zum Beispiel.

Vorgehensweise:

- Ein temporäres Objekt (der benötigten Klasse) erzeugen.
- Werte des nicht passenden Datentyps in die Klassendatentypen umwandeln.
- Das temporäre Objekt mit diesen Werten initialisieren.
- Dieses Objekt z. B. an die aufrufende Funktion übergeben.
- Löschen des temp. Objektes wenn es nicht mehr benötigt wird.

Ein Klasse Uhr besitzt die Datentypen int für Minuten und Stunden. Es soll jedoch möglich sein die Zeit als einen String einzutregen in der Form: „17:37“

```
#include <iomanip>    // wegen setw
class uhr             // Version ohne benutzererstellten Konstruktor
{
int std, min;

    public:
// Standardkonstruktor mit default-Werten
        uhr(int s = 0, int m = 0){std = s; min = m;}
// Konvertierungskonstruktor (char -> int)
        uhr(const char *zeit){

// Da die in char die Zeichen 0-9 nicht ihren Werten entsprechen
muss man sie erst in Zahlen umwandeln (7 (char) != 7(int))

//Die Abstände zwischen den Zeichen entsprechen jedoch den Ziffern,
also zwischen char 7 und char 0 liegen 7 stellen, wie auch bei den
Ziffern.

    std = 10*(zeit[0]-'0') + zeit[1]-'0';      // Da die Zeit
```

```
zweistellig ist muss das erstes Diget in zehner-Ziffer umgewandelt
werden.
    min = 10*(zeit[3]-'0') + zeit[4]-'0';        // ---//---
    }

     void show() { cout  << setw(2) << setfill('0') << std << ":"
                         << setw(2) << setfill('0') << min << endl; }
};

int main(){
// Aufruf der Konstruktoren
    uhr a;              // Ohne  Prameter uhr a(); geht nicht!!!
    uhr b(13, 17);      // Expliziter aufruf
    uhr c("09:57");     // Konvertierungskonstruktor wird aufgerufen
    uhr d ="07:36";     // Zuweisung! Konvertierung mittles Konv.-Konst

    a.show();   b.show();  c.show();  d.show();
    return 0;
}
```

Objektarrays

Zur Erzeugung und Initialisierung eines Array aus Objekten wird für jedes Element ein Konstruktor aufgerufen.

Merkmale:

- Falls alle Elemente mit dem Standardkonstruktor erzeugt und inirialisiert werden sollen kann die initiealisierungsliste entfallen: *Klasse array_objekt [Anzahl];*
- Falls das Element mit nur einem Parameter erzeugt wird, kann das Parameter direkt (ohne Konstruktoraufruf) in die Liste der Elemente eingetragen werden.
- Bei mehr als einem Parameter ist ein expliziter Aufruf erforderlich.

Analog zu dem obigen Programm kann man die Objekte a-d a auch in einem Array zusammenfassen:

BEACHTE den Unterschied bei der Erzeugung der Element-Objekte und Objekte

```
uhr ar[4] = {   uhr(),          // Initialisierung der Elemente dazu
                uhr(13,17),     wird immer der Konstruktor wie eine
                uhr("09:57"),   Funktion Aufgerufen uhr ()
                "07:36"};
```

Destruktor

Elementarfunktion deren Aufgabe es ist das Objekt wieder zu löschen und den Reservierten Speicher freizugeben.

Immer nur einer pro Klasse und erhält keine Parameter. Der Destruktor wird in der Regel immer Implizit durch den Compiler aufgerufen.

```
   class A{
public:
   ~A();            // Deklaration Destruktor
   ~A(){//Destruktoranweisungen}   // Definition innerhalb der Klasse
};
A::~A(){            // Definition Destruktor außerhalb der Klasse
                    // Hier könnten "Aufräumarbeiten" ausgeführt werden
}
```

Da der Destruktor nur die Datenelemente löscht nicht jedoch den dynamisch allokierten Heap-Speicherplatz freigibt muss man für solche Fälle selbst einen Destruktor definieren.

Implizit wird der Destruktor aufgerufen wenn:

- Objekte der Speicherklasse *auto.* Das programm verlässt den Gültigkeitsbereich (Block).
- Objekte der Speicherklasse *static* (lokal und Global). Beim Programmende.
- Mit *new* erzeugtes dynamische Objekt. Wenn Operator *delete* auf den Zeiger angewendet wird.

Methodenaufrufe sind für constatne Objekte nicht möglich. Nur wenn die Methode selbst als „const“ gekennzeichnet ist, auch bei der Definition.

Elementzeiger = Zeiger auf ein Datenelement, oder Methode

```
class Auto{
// ...
    void info()const;          // mit „const“ kann man auch konstante Objekte zugreifen
    bool fahren(int km);       // Kann nicht auf const-Objekte zugreifen!
    void tanken(float liter);
// ...
};
// ================================
// Mann kann eine Methode mit der const-Methode überladen, immer wenn ein const Objekt übergeben wird, wird die const-Methode aufgerufen, sonst die „normale“.
class A{
public:
    void methode();       // Eine Methode
    void methode()const;  // Die überladene Version  der Methode für konstante Objekte
};
```

Zugriff auf Klassenelemente

Klassenelemente sind lokale Elemente und ihr Gültigkeitsbereich ist die Klasse, in der sie Spezifiziert sind (class-scope)

Es gibt **zwei möglichkeiten** auf das Element zuzugreifen:

1. „normal“ wie auf eine normale Variable, oder Funktion (. Operator)
2. Per Zeiger. (-> Operator)

Punkt-Operator .

- Objektname . Datenelement
- Objektname . Methode

mensch.alter(31)

Der Zugrif auf die Datenelemente ist nur möglich wenn diese als „public“ deklariert wurden.
(Per default sind alle Elemente „private“)
Das gilt auch für die Objekte der selben Klasse!
Eine Methode des Objektes A der Klasse K, hat keinen Zugriff auf die privaten Elemente des Objektes B derselben Klasse.

Man sollte auf die Datenelemente immer mit public-Methoden (Schnittstellen) zugreifen. So bleiben die Datenelemente privat und können nicht direkt verändert werden.

```
class A{
private:          // !!!
    int x;
};

class B{
    A d;
    void show(){
      cout << d.x;      // Zugriif nicht möglich, da x ein privates Element der Klasse A
    }
};
```

Zugriff auf Objektarrays:

- objektname[index] . datenelement

  tank[1].wasserstand = 10;

Zugriff auf Elementobjekte

Ist ein Datenelement einer Klasse K1 selbst wieder ein Objekt der Klasse K2.

```
class dimension{
    public:
```

```
    int y;      // Anlegen eines Datenobjektes der Klasse dimension
    double lenght, width, height;
};

class tank{
    public:
    dimension d;      //Erzeugen eines Objekts d der Klasse dimension

    void show(){
    cout <<d.x;
        }
    };

int main(){
    tank a;           // Ein Objekt a der Klasse tank erzeugen
    a.d.lenght = 10;       // Zugriff auf das Datenelement lenght
                      der Klasse dimension mit dem Objekt a der
                      Klasse tank.
// Zugriff auf die Variable lenght, des Datentyps dimension im
Objekt a der Klasse tank.

return 0;
}
```

Pfeil-Operator ->

Der Pfeil-Operator wird verwendet wenn der Zugriff auf einen Klassenobjekt über einen Zeiger erfolgt – nicht über den Objektnamen! Wenn man z. B. ein Objekt an eine Funktion übergibt, oder wenn ein Objekt dynamisch auf dem HEAP angelegt wird.

Zeiger->Element = (*Zeiger).Element

```
class mensch{
// --- Klassendefinition ---
double lenght;
};

//Zeiger auf Objekt

//Nicht Dynamisch Erzeugtes Objekt.
mensch Mann;               // Nicht dynamisch erzeugtes Objekt Mann
mensch *z = &Mann;         // Zeiger z auf das Objekt

//Dynamisch Erzeugtes Objekt.
Zeiger zeigt auf ein Datenelement innerhalb der Klasse, die
Adresse(Offset) ist für alle Objekte gleich!

mensch *z = new mensch;    // Speicher für ein Objektder Klasse
                           mensch wird allokiert und  die Adresse an
                           Zeiger z übergeben.

// Vergleiche dazu:        (int *z = new int;)
    int *z;                // Zeiger auf ein int-Wert definieren
    z = new int;       // Speicherplatz für einen int-Wert allozieren
                      und die Adresse dem Zeiger z zuweisen.
```

```
// Zugriff erfolgt über:

(*z).lenght = 7.7;    // Die Klammern muss man verwenden, da die
                      Priorität . > *
z->lenght = 7,7;      // Das Gleiche mit dem Pfeil-Operator
```

Elementenzeiger

`Zeiger auf Elemente eines bestimmten Typs einer Klasse.`
Zeiger erhält nicht die Adresse, sondern den Offset zur Anfangsadresse des Objektes in Bytes, dieser ist für alle Objekte gleich.
Die Adressierung erfolg dadurch, dass die Adresse des Objektes genommen wird und dazu der Offset des Datenelementes hinzuadiert wird.

Die Elememtauswahloperatoren sind: `.*` und `->*`

Normaler Zugriff:
- Adresse_Objekt . Element
- Adresse_Objekt -> Element

Zugriff über Elementzeiger (Offset):
- Adresse_Objekt .* Offset_Element (Elementzeiger)
- Adresse_Objekt **->*** Offset_Element (Elementzeiger)

Gewöhnlicher Zeiger:

Der Unterschied und Vorteil von Elementzeigern ist es, dass dieser keinen konkreten Objekt erfordert! Im unterschied zu gewönlichen Zeigern.

Gewöhnliche Zeiger und Elementen Zeiger sind nicht kompatibel!

```
//***Zeiger auf ein Datenelement eines bestimmten Typs***

class Mensch{
public:
    double groesse_;    // 1. Eintrag double = 8 Byte
    double gewicht_;     // 2. Eintrag der Offcet = 8 Byte
    // ---//---
};

int main(){

//dz = double Zeiger, Zeiger auf Datentyp double
double Mensch::*dz;    // Deklaration: Zeiger auf (nur) Datenelemente
                       des Typs double der Klasse Mensch

dz = &Mensch::gewicht_;     // Dem Zeiger das Datenelement Gewicht zuweisen.

//(double Mensch:: *dz = &Mensch::gewicht_;)
Mensch frau, mann;  // Zwei objekte der Klasse Mensch erzeugen

// Zugriff:
```

```
//   (objekt.*Elementenzeiger;)

mann.*dz = 80;              // Zugriff auf das Gewicht über den
                            Zeiger auf das Datenelement
mann.gewicht_=80;           // Zugriff über den Datenelementnamen
frau.*dz = 60;              // Der Zeiger gilt für alle Objekte dieser Klasse

//********* wenn keine Objekt vorliegt ************
// Ohne Objekt wird der Pfeil-Operator verwendet
//oz = Objektzeiger, Zeiger auf ein Objekt der Klasse

// (Zeiger_auf_Objekt -> *Elementzeiger;)

Mensch *oz = new Mensch;          //Neues (unbenanntes) Objekt auf dem
                                  Heap erzeugen, allokieren
//Zugriff

oz ->* dz = 70;    // Einem "abstrakten" Objekt wird das Gewicht(dz) 70 zugeordnet
oz -> gewicht_ = 70;        // Mann kann auch direkt auf das gewicht zugreifen

//Zugriff mit gewöhnlichen Zeigern
    double *p;
    Mensch peter;
    p = &peter.gewicht_;
    *p = 75;
return 0;
}
```

Zeiger auf Elementenfunktionen

Zugriff auf (ausschließlich) Elementenfunktionen eines bestimmten Typs.

```
1. Typ_Rückgabewert (Klassenname::*Zeigername) (Parameterlsite);
2. Zeiername = Klassenname::Funktionsname;

Typ_Rückgabewert (Klassenname::*Zeigername) (Parameterlsite) =
Klassenname::Funktionsname;
          void (Mensch::*zef)(double) = Mensch::f;
```

(siehe Zeiger auf Funktionen: int f(double d); => int (*fz)(double d);)

Mensch::f (Klassenname::Funktionsname) = qualifizierte Name der Funktion ist ein konstanter Zeiger auf die Funktion (kein Offset). Also die Adresse im Codesegment.

Der Zeiger einer Elementfunktion ist mit dem Zeiger auf eine gewöhnlichen Funktion inkompatibel! `Void (*zf)(double) = Mensch::f; FEHLER!!!`
Weil die Elementfunktionen als zusätzlichen Parameter den This-Zeiger enthalten.

```
Zugriff auf die Funktion:
Linker Operand = Objekt                  => .*
Linker Operand = Zeiger auf Objekt       => ->*
```

```
(objekt.*elementzeiger) (parameterliste)
(Zeiger_auf_Objekt->*elementzeiger) (parameterliste)
```

```
class Mensch{
public:
    double groesse_;    // 1. Eintrag double = 8 Byte
    double gewicht_;     // 2. Eintrag der Offcet = 8 Byte
    int alter;
    // ---//---
    void setalter(int new_alter) {
    alter=new_alter;
    }
};

int main(){
    // void (Mensch::*zef)(int new_alter);
    //  zef=Mensch::setalter;

void (Mensch::*zef)(int)=&Mensch::setalter;      // & nötig - Fehler
                                                 im Buch?
    Mensch peter;                // Erzeugt ein neues Objekt
    Mensch *zo = new Mensch;     // Allokiert Speicher für eine
                                 Objekt der Klasse Mensch

//Zugriff:
    (peter.*zef)(25); // = peter.setalter(25);
    (zo->*zef)(27);     // zo->setalter(27);
    cout << peter.alter << endl;      // Datenelement eines Objektes ausgeben
    cout << zo->alter << endl;        // Datenelement eines Zeigers
                                      auf ein Objekt ausgeben
    return 0;
}
```

This-Zeiger

Alle Datenelemente sind für jedes Objekt einer Klasse separat angelegt, mit der Ausnahme der static-Datenelemente, sie sind wie die Elementenfunktion nur ein Mal vorhanden.

Damit die Elementfunktionen auf die Datenelemente des Objekts zugreift für das sie aufgerufen wurde wird ein versteckter Parameter an die Funktion mitübergeben. Der This-Zeiger enthält die Adresse des Objekts, mit der er automatisch initialisisert wird, sobald die Funktion aufgerufen wurde.

This-Zeiger = konstanter Zeiger auf die Objekte der Klasse.

void set_alter (**Mensch *const this**, int new_alter){---//---} //der Zeiger ist verschteckt

In der realität greift man so auf ein Datenelement zu mit: this->alter
So wandelt der Complier die Aufrufe der Elementfunktionen um:

```
  Mensch peter;             // Erzeugt ein neues Objekt
  Mensch *zo = new Mensch; // Adresse eine virtuelen Objektes

- peter.set_alter(25);      =>    set_alter(&peter, 25);
- zo->set_alter(27);        =>    set_alter(zo, 25);
```

Der große Vorteil des This-zeigers liegt in der **Verkettung von Aufrufen** von Elementfunktionen.
Durch die Referenz als Rückgabewert wird der Funktionsaufruf zum L-Wert.

Gibt eine Funktion sich selbst zurück: return(*this) kann der Aufruf einer zweiten Funktion einfach drangehängt werden: (*this).get_y() = pget_y()

```
class Coordinate
{
    int xcoor, ycoor;
public:
    Coordinate get_x(){    // Einlesen x-Koordinate
        cin >> xcoor;
        return(*this);}    // Rückgabe (*this) = p
    Coordinate get_y(){    // Einlesen y-Koordinate
        cin >> ycoor;
        return(*this);}    // Rückgabe (*this) = p
    void show_xy(){        // Ausgabe
        cout << xcoor << " " << ycoor << endl;
    }
};

int main(){
    Coordinate p;     // Ein Objekt erzeugen (p = Punkt)
    p.get_x();        // 3 Funktionen der Reie nach aufrufen
    p.get_y();
    p.show_xy();
// Weil return(*this) das Objekt selbs zurückliefert heiß die
Rückgabe von p.get_x() = p, das bedeuted für den Zweiten Teil, das
Selbe wie p.get_y();
    p.get_x().get_y().show_xy();      // drei Funktionsaufrufe mit
                                      einer Anweisung
    return 0;
}
```

Konstante Objekte

Wenn das Objekt als *const* deklariert ist sind alle Datenelemente des Objekts Konstant.
Das konstante Objekt muss gleich beid der Erzeugug initialisiert werden.

```
const K a(7,13);
const K b;       //Fehler! Das const-Objekt muss initialisiert werden
```

Der Compiler lässt auch den Aufruf der Methoden nich zu, weil er nicht feststellen kann ob diese die Datenelemente verändert.
Der Aufruf ist nur dann gestattet, wenn die Methode selbst als const gekennzeichnet ist.

Mit dem schüsselwort ***mutable*** lassen sich einezelne Datenelemente in den nicht konstanten Zustand überführen.

Konstante Methoden

Die konstante Elementfunktion kann keine Datenelemente verändern, deshalb lässt sich mit ihr auch auf die konstanten Objekte zugreifen.

```
void f() sonst{}          // Definition: Schlüsselwort const hinter den runden Klammern
                          bei Definition und Deklaration
```

Der Datentyp des this-Zeigers entscheidet über die Zugriffsfähigkeit der Methode.

Normale Methode:	K* const	= Konstanter Zeiger
Konstante Methode	const K* const	= Konstanter Zeiger auf konstantes Objekt

Statische Klassenelemente

Statische Datenelemente

Merkmale:
- Lebensdauer bis zum Programmende (wie globale Variablen).
- Gehört nicht zu einem Objekt sondern der ganzen Klasse. So wird es von allen Objekten gemeinsam genutzt.
- Der Gültigkeitsbereich im Unterschied zur globalen Variablen ist nur die Klasse.

Es sind keine statischen Elemente in Lokalen klassen Erlaubt.

Statisches Datenelement erzeugen:
1. Innerhalb der Klasse mit dem Schlüsselwort *static* deklarieren.
2. Ohne static, mit dem qualifiziertem Namen (*::*) außerhalb der Klasse definieren.

Der Zugriff erfolgt wie auf nicht statische Datenelementen.

```
class K{
    public:
    static int a;           // Deklarieren
};

int K::a;        // Definieren (hier static verboten!)
                 // Weil static-Var. implizite Initialisierung mit 0.

int main(){
    K::a =5;           // gewöhnlicher Zugriff

    K obj;
    obj.a = 6;         // Zugriff über das Objekt „obj“

    K *zo = &obj;      // Dem Objektzeiger das Objekt „obj“ zuweisen
    zo->a = 7;         // Zugriff über den Objektzeiger „oz“

// Beim Zugriff ist es völlig egal welches Objekt man verwendet, da
static Variable nicht zum Objekt sondetn zur Klasse gehört

return 0;
}
```

Statische Methoden

Der Anwendungsbereich der statischen Methoden beschränkt sich auf den Zugriff auf die statischen Datenelemente, das sie dabei efizienter sind als die nicht statischen Methoden.

Die statischen Methoden steht kein this-Zeiger zur Verfügung, deshalb ist der Zugriff auf die nicht statischen Datenelemente verboten.

Der this-Zeiger kennzeichnet ein Objekt auf die sich die Funktion bezieht, da die nicht statischen Datenelemente einem Objekt gehören kann ohne this-Zeiger kein Objekt und somit kein Datenelement ausgewählt werden.

Um dennoch auf die nicht statischen Datenelemente zugreifen zu können benötigt die statische Methode eine Objekt, oder einen Zeiger darauf.

```
class K{
    static int a;
    int b;
public:
    K(int pb=0){b=pb;}     // Defaultkonstr. (pb = Parameter für b)

    static void f1(){
        cout << a << endl;;      // Erlaubt
   //   cout << b;          // kein Zugriff auf die nicht statischen
                            Datenelemente
    }
    static void f2(K *z);  // Zeiger auf ein Objekt der Klasse K
};
int K::a = 7;         // Definition des statischen Datenelements

// Zugriff auf nichtstatisches Datenelemnt ist nur über ein Objekt
oder einen Zeiger zulässig.

    void K::f2(K *z){           // Zeiger z auf Objekt als Parameter
    cout << z->b << endl;       //
}

int main(){
//************** Definition von Objekten ***************
    K obj = 3, *op;   // Objekt anlegen und b den Wert 3 zuweisen.

    K *p;        // Zeiger anlegen
    p = &obj;    // Zeiger verweist auf dad Objekt obj.
    // K *p = &obj;        Zusammengefasst

//*************** Zugriff **************

    K::f1();     // Aufruf der static-Methode
    K::f2(p);    // Zugriff auf die nicht statischen Datenelement b
                 über Zeiger, oder Objekte

    K::f1();          // Direkter Aufruf
    op->f1();         // Aufruf über den Objekt-Zeiger op
    obj.f1();         // Aufruf über das Objekt

return 0;
}
```

Friend-Funktionen

Um einer Funktion den Zugriff auf die privaten Datenelemente einer Klasse K zu gestatten muss die Funktion als *friend* der Klasse deklariert werden. Dabei kann es sich um eine Globale Funktion, oder eine Methode der anderen Klasse handeln.

Definition Klasse B, in der eine Methode als friend deklariert wird, muss NACH der Klasse A erfolgen wo diese Methode definiert wird. Erst Deklaration der Methode, dann die friend-Deklaration!

Da, die Methode nicht vor ihrer Klasse deklariert werden kann. Dieser Umstand mach es unmöglich das Methoden der Klassen A und B den gegenseitigen Zugriff auf die Datenelemente haben. Dies kann man mit der Deklaration der friend-Klassen umgehen.

Es ist egal in welchen Zugriffsbereich (private, public, ...) man diese deklaration durchführt.

Die Funktion wird dabei nicht zur Methode der Klasse, das heißt sie verfügt nicht über den this-Zeiger als Parameter. Was wiederum heißt, dass für den Zugriff auf die Datenelemente der Klasse sie den entsprechenden Zeiger auf das Objekt benötigt.

```
class B;    // Dekl. der Klass B, weil sie in A verwendet wird

class A{
public:
    void show_M(B ob_B);
};
//*********************
class B{
    int var=7;

public:
    friend void A::show_M(B ob_B);     // Methode der Klasse A als
friend deklarieren
    friend void show_GF(const B&);      // Glob-Funktion show_GF als
firend der Klasse B deklarieren
};

// ********************** Funktionsdeklaration ******************

void show_GF(const B& ob_B){     // Def. glob-Funktion mit dem
                                 konstantem Parameter Referenz auf
                                 Objekt "ob_B" der Klasse B
ob_B.var++;                   // wegen const ist kann das Objekt nicht
                              modifiziert werden
    cout << ob_B.var << endl;
}

void A::show_M(B ob_B){          // Def. Methode der Klasse A mit
                                 Parameter Objekt "ob_B" der Klasse B
ob_B.var++;           // Da das Objekt nicht als const übergeben
                      wurde kann "var" verändert werden
cout << ob_B.var << endl;
}
int main(){ //*************************************************
    B obj_B;         // Objekt der Klasse B erzeugen
    A obj_A;

    show_GF(obj_B);         // Globale Funktion mit dem Objekt
                            "obj_B" als Argument aufrufen
    obj_A.show_M(obj_B);

    return 0;
}
```

Sollen alle Methoden der Klasse B auf die Elementer der Klasse A zugreifen können, kann man die Klasse B als *friend*-Klasse der Klasse A derklarieren.

Im Gegensatz zu friend-Funktionen spielt die Reihenfolge der Klassendefinitionen bei friend-Klassen keine Rolle! Weil eine Klasse vor der Definition deklariert werden kann.

```
class A{

    friend class B;
};

class B{

    friend class A;
   };
```

Abgeleitete Klassen

Sämtliche Elemente werden and die abgeleitete Klasse von der Basisklasse übergeben.
Nicht übergeben werden nur Konstruktoren, Destruktoren und Zuwesungsoperatoren.
Ziel: Erweiterung/ Spezialisierung der Basisklasse

```
class Abgeleitete_Klasse : Zugrifsspezifizierer[public, private,
protected] Basisklasse_1, Zugrifsspezifizierer_2 Basisklasse_2, ...
```

- Basisklasse muss vorab definiert werden
- Zugrifsspezifizierer per default = *private*

private: Nur Methoden der Klasse und *friends* [**nicht für abgeleitete Klassen**]
protected: Methoden der Klasse, *friends* und abgeleitete Klassen
public: Keinerlei Zugriffsbeschränkungen für das Klassenelement

Speicherbereich der Abgeleiteten Klasse:

Elemente der Basisklasse				Elemente der abgeleiteten Klasse	
(Vptr)	X	Y	Z	A	B
O = Objekt der abgeleiteten Klasse					

(vptr, siehe virtuelle Funktionen)
Buch Seite 813.

Basisclasse	Arte der Ableitung	Abgeleitete Klasse
private	**--- : public --->**	Kein Zugriff!
private	**--- : protected --->**	Kein Zugriff!
public		*protected*
private	**--- : private --->**	Kein Zugriff!
public		*private*
protected		*private*

```
class B{
  public:
    int x;
  protected:
    int y;
  private:
    int z;
};
```

Public Ableitungen:

```
class A : public B {...}; // A = Abgeleitet, B = Basis
```

Der Zugrifsspezifizierer Private der Basisklasse wird in der abgeleiteten klasse verändert.

```
//*******Abgeleitete Klasse*******
class A : public B {        // A = Abgeleitet, B = Basis
int a;
Hier besitzt die abgeleitete Klasse ein Element „z“ auf die sie keinen Zugriff hat, da private!

int f(){
return a;  // die Methode der Klasse hat Zugriff alle eigenen Elemente
return x;             // OK! da in der Basisklasse public
return y;             // OK! Da in B protected (hier auch)
                             somit is innerhalb der Klasse sichtbar
return z;  // Fehler! Da „z“ in der Basisklasse private können nur friends darauf zugreifen
    }
};

int main(){
    f();
}
int f(){
    A::A objA;        // das Selbe gilt für objekt B::B objB;

    objA.a =3;        // Fehler! Kein Zugriff da private!
    objA.x =7;        // OK! da public!
    objA.y =13;       // Fehler! Kein Zugriff da protected!
    objA.z =17;       // Fehler! Kein Zugriff da private!
}
```

protected Ableitungen

```
class A : protected B {...}; // A = Abgeleitet, B = Basis
```

```
//******* Abgeleitete Klasse *******
class A : protected B {         // A = Abgeleitet, B = Basis

// „x“ = protected. Durch protected vererbung werden public Elemente
der Basisklasse zu protected Elementen in der Abgeleiteten Klasse

    int f(){
```

```
        return x;     // OK! in der Basisklasse public hier protected
        return y;    // OK! Da in B protected (hier auch)
        return z;     // Fehler! Da z private in der Basisklasse
    }
};

int main(){
    B objB;
    A objA;
    objB.x =3;  // OK! In B public
    objA.x =7;  // Fehler! Da in A protected
   }
```

private Ableitungen

```
class A : private B {...}; // A = Abgeleitet, B = Basis
```

```
//*******Abgeleitete Klasse*******
class A : protected B {        // A = Abgeleitet, B = Basis
// Bei einer privete Ableitung werden public und protected Elemente
„x“ und „y“ in der abgeleiteten Klasse zu private!
    int f(){
        return x;       // OK! in B public hier protected
        return y;       // OK! in B protected hier private
        return z;       // Fehler! Da z private in der Basisklasse
    }
};

int main(){
    B objB;
    A objA;

    objB.x =3;        // OK! public
    objB.y =7;        // Fehler! protected
    objB.z =13;       // Fehler! private

    objA.x =3;        // Fehler! private in A
    objA.y =7;        // Fehler! private in A
    objA.z =13;       // Fehler! kein Zugriff in A
   }
```

Redifinition der Zugriffsrechte (Wiederherstellen)

Es lassen sich nur Zugriffsrechte von *public* und *protected* Elementen redefinieren, da man auf private ja keinen Zugriff hat.

Die Zugriffsrechte kann man nur wiederherstellen nicht neu vergeben, sie können nicht erweiter oder beschränkt werden.

```
class B{
public:
    int x;
    void f(){cout << "Hallo"<< endl;};
protected:
    int y;
```

```
private:
    int z;
};
```

```
//*******Abgeleitete Klasse*******
class A : protected B {        // A = Abgeleitet, B = Basis

public:
    B::x;       // x wird wieder public
    B::y;       // Fehler! Rechte dürfen nicht erweitert werden
    B::f;       // Bei der Funktion wird nur der Name angegeben
                ohne ()!
protected:
    B::x;       // Fehler! Rechte dürfen nicht eingeschränkt werden
    B::y;       // y wird wieder protected
};
```

Redifinition von Klassenelementen

- Die Namen der Elemente in Basisklasse können in der abgeleiteten Klasse überdeckt werden, so wie globale Varablen von den lokalen überdeckt werden.
Überdecken ≠ Übeladen!
- Beim überdecken existieren alle Elemente und Funktionen.
- Man kann auch einen anderen Datentyp wählen, nicht jedoch die Zugrifsrechte verändern.

Auf die Elemente der der Basisklasse in qualifizierter Form (::) zugreifen.

- **Qualifizierter Name** = Klasse::Element
- objA.f() => objA.B::f(); // B = Basisklasse
- Es ist egal ob es die Methode (oder Variable) aus der Direkten, oder indirekten Basisklasse abgeleitet wurden, der Qualifizierte name bleibt gleich:
- B -> A -> K -> U Der Zugriff mit dem Objekt (U objU) auf die Funktion f() der Klasse B erfolgt durch: objU.B::f()

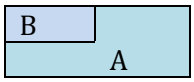
```
class B{
public:
    int x;
    int y;

    void f(){cout << "Funktion der Basisklasse B"<< endl;};
};
//*******Abgeleitete Klasse*******
class A : public B {        // A = Abgeleitet, B = Basis

public:
    int x;
    double y;

    void f(){cout << "Funktion der abgeleiteten Klasse A"<< endl;};
};

int main(){
    //B objB;
    A objA;
```

```
    objA.x =3;
    objA.y =7.7;

    objA.B::x = 13;   // Zugriff auf die überdeckte Variable
    objA.B::y = 17;   // der Basisklasse

    cout << "A.x:" << objA.x <<" B.x:" << objA.B::x << endl;
    cout << "A.y:" << objA.y <<" B.y:" << objA.B::y << endl;

    objA.f();
    objA.B::f();      //Zugriff auf die „originale“, überdeckte Funktion der Basisklasse
}
```

In der Regel werden Datenelemente nicht überdeckt, Methoden hingegen schon, siehe Virtuele Funktionen.

Implizite Konvertierungen zwischen den Erb-Klassen

- Besteht zwischen den Klassen ein ***public***-Ableitungsverhältnis, werden die Objekte/Zeiger/Referenzen bei bedarf vom kompiler implizit konvertiert.
- Die Konvertierung ist jedoch immer nur in eine Richtung möglich:
  B ---*public*---> A
  (Objekte/Zeiger/Referenzen)von A in → (Objekte/Zeiger/Referenzen) von B
  jedoch nicht andersrum!
- Objekte der abgeleiteten Klassen dürfen nicht mit Objekten der Basisklassen initialisiert werden.
- Das Kommt daher, dass die Klasse B eine Teilklasse von A ist, das heißt, dass alle Elemente von B auch in A vorhanden sind. Jedoch sind nicht alle Elemente von A in B vorhanden.




```
class B{
};

class A : public B {         // A = Abgeleitet, B = Basis
};

class K : private B {
};

void fB_1(B x);      // Parameter x der Klasse B
void fB_2(B* x);     // Parameter: Zeiger x auf den Typ Klasse B
void fA_1(A x);       // erwartet parameter des Typs A
void fA_2(A* x);      // erwartet einen Zeiger auf den Typ A

int main(){
    B objB;
    A objA;
    K objK;

    fB_1(objA);       // implizite Konvertierung vom Objekt/Zeiger
    fB_2(&objA);      // der Klasse A in das Objekt der Klasse B

    fA_1(objB);       //Fehler! Konvertierung aus der Basisklasse in
    fA_2(&objB);      // die abgeleitete Klasse nicht möglich
```

```
    fB_1(objK);       //Fehler! B → K Ableitung nicht public -> kein
}                     // Zugriff auf die Elemente von B aus K
```

Es kann keine Adresse eines Elements der Basisklasse einem Zeiger der abgeleiteten Klasse zugewiesen werden.

- Zeiger Zeigen nur auf Elemente des ihres Typs!
- Die abgeleitete Klasse A enthält alle Elemente des Typs der Basisklasse B.
- Ein Zeiger des Typs B, auf ein Objekt des Typs A zeigt nur auf die Elemente des Typs B!
- Somit gibt es keine Überschreitung des Objektpeichers

- Mann kann jedoch mit einer expliziten Konvertierung des Zeigers des Typs B auf die (nicht vererbte) Funktion der abgeleiteten Klasse A zugreifen. `((A*)zB)->f(A);`
- Es gibt eine Möglichkeit ohne Konvertierung auf die Elemente der abgeleiteten Klasse mit dem Basisklassen-Zeiger zuzugreifen, siehe *Virtuele Funktionen.*

```
class B{
public:
    int a;
    void f(){ cout<< "f von B" << endl;}
    B(): a (7){}        // Konstruktor mit initialisierung von a
};
class A : public B {        // A = Abgeleitet, B = Basis
public:
    void f(){cout<< "f von A" << endl;}
    void g(){cout<< "g von A" << endl;}
};

int main(){
    B objB;
    A objA;

    objB = objA;      // OK! Ableitung des Objektesder Klasse A aus
    objA = objB;      // Fehler! B enthält nich alle Elemente von A

    A *zA = &objA;    // Zeiger auf das Objekt "objA" des Typs A
    B *zB = zA;       // OK! Implizite Konvertierung des Zeigers vom
                      Typ A in den Zeigr vom Typ B

    B *zB_2 = &objB;       //Zeiger auf das Objekt "objB" des Typs B
    A *zA_2 = zB_2;        // Fehler! Konvertierung der Adresse der
                           Oberklasse B in die Unterklasse A

cout << zB->a;    // Zeiger auf das Objekt der Klasse A dieser kann
cout << objA.B::a << endl  auf objekte der Klasse B zugreifen
                           (implizite Konvertierung) => B::x


    zB->f();         // B::f =>      objA.B::f(); f von B
    zA->f();         // A::f =>      objA.f();  f von A

zB->g();        // Fehler! Der Zeiger ist vom Typ B und obwohl er auf
                 das Objekt von A zeigt, sieht er nur Elemente von B
((A*)zB)->g(); // Zugriff nur durch explizite Konvertierung möglich
}
```

Nicht vererbbare Klassenelemente

Friends

- Freund-Funktionen, oder Freund-Klassen können nur auf die Freundklasse und die *public*-Elemente der Erbklasse zugreifen, nicht jedoch auf Elemente der A klasse die *private* oder *protected* sind.
- Will man auf alle Elemente der abgeleiteten Klasse A zugreifen, so wird in der Klasse A auch die *friend*-Deklaration eingefügt: `friend void f_fr(); friend class A;`
- Die „Freindschaften" werden nicht vererbt! Ist die klasse B Freund von A, und die Klasse C Freund von B, so ist C immer noch kein Freund A!
- A (*friend*)→ B (*friend*)→ C A ~~*friend*~~ C

Konstruktoren

- Konstruktoren dürfen daher nicht abgeleitet werden, da die sie nicht alle Elemente der abgeleiteten Klasse initialisieren können.
- Daher wird die Arbeit geteilt. Zur initialisierung der abgeleiteten klasse wird der Konstruktor der Basisklasse aufgerufen und der eigene Konstruktor, für die neu dazugekommenen Elemente.
- Der Basisklassen-Konstruktor wird in der Initialisierungsliste des Klassenkonstruktors (abgeleiteten Klasse) aufgerufen.

Die abgeleiteten Elemente dürfen in der Initialisierungsliste nicht reinitialisiert werden.

```
class B{
public:
    int x;
    B(int a_neu=0): a (a_neu){} //Konstruktor B (mi tder Initialisierung von a)
};

class A : public B {
public:
    int y;
    double z;
// Aufruf des Konstruktors A in dem einegene Objekte initialisiert werden und der Konstruktor der Basisklasse aufgerufen wird:
    A(): B(6), y (3), z (3.7) {}
};
```

```
// B wie oben

class A : public B {         // A = Abgeleitet, B = Basis
public:
    int b;
    double c;
    A(int B_neu=0, int b_neu=0, double c_neu=0 ):
                                B(B_neu), b (b_neu), c (c_neu) {}
};
int main(){
    A oA(3,6,7.9);          // a=3, b=6, c=7.9
   }
```

Der Standartkonstruktor der Basisklasse mus nicht explicit aufgerufen werden, er wird mit der Erzeugung des Objekts implizit aufgerufen.

```
// B wie oben

class A : public B {         // A = Abgeleitet, B = Basis
public:
    int b;
    double c;
A(int B_neu=0, int b_neu=0, double c_neu=0 ):b (b_neu), c (c_neu) {}
};
int main(){
    A oA(3,6,7.9);          // a=0,(die 3 wird nicht berücksichtigt)
                            b=6, c=7.9
   }
```

Reihenfolge der Konstruktoraufrufe:

1. Basisklassen
2. Elementobjekt-Klassen // Objekt der Klasse X innerhalb der Klasse Y erzeugen
3. Abgeleitete Klassen

Werden nicht in der Reihenfolge aufgerufen, in der sie in die Initialisierungsliste geschrieben sind!

```
class B{
public:
    int x;
    B(int x_neu=0): x (x_neu){}       // 1. Basisklassenkonstruktor
};

class A : public B {         // A = Abgeleitet, B = Basis
public:
     int y;
     B objB;          // 2. Elementobjekt-Konstruktor

A(int B_init=0,int b=0,int y_neu=0): objB(b), B(B_init), y(y_neu) {}
// 3. Konsturktor abgeleitete Klasse
};

int main(){
    A oA(3,6,7);
    cout << oA.x<< "\n" << oA.y << "\n";   // 3 7
    cout << oA.objB.x << endl;             // 6  objB.x geht nicht!
}
```

Destruktoraufrufe

Die Destruktoren werden in der follgenden Reihenfolge aufgerufen.

1. Ableitungsdestruktor
2. Elementendestruktor
3. Basisklassendestruktor

Virtuelle Funktionen *VF*

- Eine Klasse die eine virtuelle Funktion besitzt bezeichnet man als **polymorphe Klasse**.
- Sehr nützlih wenn die mehrere Kinder einer Klasse dieselbe Funktion besitzen (z.B. erzeuge, lösche, ...) So kann ma mit dem Zeiger des gleichen Typs, die Funktion des jeweiligen Objekts aufrufen.
- Mit einer Virtuellen Funktion kann man mit einem Basisklassenzeiger auf Funktionen der abgeleiteten Klasse zugreifen, ohne dabei eine Konvertierung des Zeigertyps durchzuführen.
- Das Objekt auf das der Zeiger verweist entscheidet darüber welche Funktion Aufgerufen wird.
- Bei nichtvirtuelen Funktion entscheidet der Typ des zeigers über die Funktion die aufgerufen wird.
- Die Funktion wird in der Basisklasse, innerhalb einer Klassendefinition, als *virtual* deklariert, oder definiert und in der abgeleiteten Klasse redefiniert.
- Bei einer Definition außerhalb der Klasse darf der Spezifizierer *virtual* nicht mehr verwendet werden.
- Bei der Redefinition müssen Parameter und der Rückgabewert exakt übereinstimmen.
- Unterscheidet sich nur der Rückgabewert gibt es einen Fehler, da eine virtuale Funktion nicht überdeckt werden kann.

```
class B{
public:
    virtual void f(char);        // Virtuale Funktionen f und g
    virtual void g(void);
};

virtual void f(char a){}       // Fehler! außerhalb der Klasse darf
                                 virtual nicht verwendet werden
// Außerhalb der Klasse Qualifizierter Name! (::)
void B::f(char x){cout << "B::f \n";}
void B::g(){ cout << "B::f \n";}

class A : public B {
public:
    int f(char y){}   // Fehler! Untersch. Rückgabewert = virtuele
                      Funktion kann nicht überdeckt werden.

     void f(char y){...}  // virtual, Parameter und R-Wert gleich
     void g(int a){...}
// Ok, aber: Parameter nicht exakt gleich, keine virtuele Funktion
=> Zugriff auf Basisklassenversion von f(), da der Typs des Zeiger
entscheidet.
 };
int main(int argc, const char * argv[]){
    A oA;

    B* zB_A =  &oA;      // Zeiger auf oA des Typs B
    A* zA_A = new A;     // Zeiger auf oA(nicht real) des Typs A

    zB_A->f('V');     //2x A::f f() = virtuale Funktion, entscheidend
    zA_A->f('V');     // das Objekt (2x Klasse A) nicht der Typ

    zB_A->g();        // B::g, g() ist nicht virtual, Zeiger-TYP entscheidend
```

```
    zB_A->g(3); // Fehler! Keine Virtuale Funktion = Zugriff mit dem
Zeiger zB auf Funktion von A nicht möglich (ohne Konvertierung)
    zA_A->g(7);       // A::g, Nur direkt über Zeiger des Typs A, mit
                      richtigen Parametern zugreifbar
}
```

Die Redefinition kann auch in einer indirekt abgeleiteten Klasse Stattfinden.

B(Definition virtual f()) → A(~~f()~~) → K(Redefinition von f())

Enthält die Klasse des Objekts oK, das für den Funktionsaufruf verantwortlich ist, keine Redifinition wird einfach die nächste drüberliegene redefinierte Funktion aufgerufen.

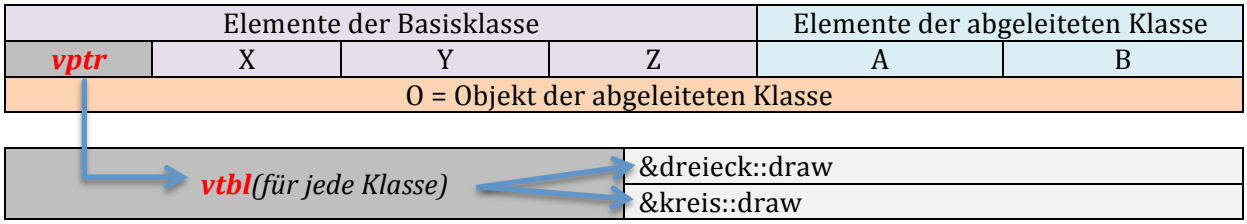
```
class B{
public:
    virtual void f();
};
void B::f(){}

class A : public B {
 public:
    void f(){cout << "A::f" << endl;}
 };

class K : public A {
public:
~~f(){}~~
};

int main(int argc, const char * argv[])
{
B* zB_K = new K;            // Da die Klasse K keine Redefinition
                            von f enhält wird A::f aufgerufen
        zB_K->f();
}
```

Man stellt sich vor, dass die Zeiger während der Programmlaufzeit generiert werden, so dass der Compiler nicht weis welche Funktion aufgerufen wird. Wie ist die Compelierung doch möglich?

Es wird für jedes Objekt (Klasse), welches virtuele Funktionen enthält, ein **Zeiger** auf eine **Tabelle**(Array) von Funktionszeigern, die Adressen der virtuelen Funktionen speichern, angelegt.

- Zeiger: virtual function pointer (vptr)
- Tabelle/Array: virtual function table (vtbl)

Der Zeiger ist verborgen und für den Programmierer nicht zugreifbar



Wird eine virtuelle Funktion aufgerufen, ermittelt das Programm (zur Laufzeit) über den vptr-Zeiger die Adresse der Funktion aus der zugehörigen vtbl und führt sie aus.

```
Symbol* zS_D = new dreieck;      // vptr[0]
zS_D->draw();          Programmintern →      zS_D->vptr[0]();
```

Konstruktor / Destruktor

-Der Konstruktor kann nicht virtuell sein.

- Um ein A-Objekt zu konstruieren muss zuvor ja das Teilobjekt B konstruirt werden.

- Enthält der Basisklassenkonstruktor eine virtuele Funktion, so zeigt seine vptr auf die vtbl, ebenfalls von B, da die anderen Objekte ja noch nicht konstruirt sind gibt es keine redefinierten Funktionen der Klasse A.

- Es wird also immer der Basisklassenkonstruktor ausgeführt.

Destruktor:

Destruktoren können dagegen virtuell sein und müssen sogar manchmal.

Will man z.B. den Zeiger *zB_A (B* zB_A = new A;)* löschen, so genügt es nicht *delete zB_A;* aufzurufen. Da der Zeiger vom Typ B ist wird nur der B-Destruktor ausgeführt. Die Zusatzelemente des A-Teils bleiben im Heap.

Destruktoren haben für das System alle den gleichen Namen, deswegen können sie virtuell sein.

Spezifiziert man den Basisklassen-Destruktor als virtuell, sind die Destruktoren seiner abgeleiteten Klassen automatisch virtuell.

Definiert man jetzt den Basisklassendestruktor *virtual ~B(){},* wird mit *delete zB_A;* erst das A-Teil gelöscht und und anschließend der Basisklassendestruktor aufgerufen.

Zugrifsstatus

Hängt von dem Typ(Klasse) des Zeigers, der auf die virtuelle Funktion verweist.

```
class B{
public:
    virtual void f();
};
void B::f(){cout << "B" << '\n';}

class A : public B {
 private:
    void f(){cout << "A";}
 };

int main(int argc, const char * argv[])
{
        A* zA_A = new A;        // Zeiger auf den Typ A
        B* zB_A = new A;        // Zeiger auf den Typ B

    zA_A -> f();           // Fehler! Da A::f private
    zB_A -> f();           // OK, da der Zeiger auf A::f zeigt und
                           es ist public
}
```

==========

std::shared_ptr ist ein intelligenter Zeiger (smart pointer), der ein Objekt über einen Zeiger besitzt. Mehrereshared_ptr Instanzen können das selbe Objekt besitzen.

Der Vorteil von Smart Pointern gegenüber normalen Zeigern ist, dass sie die Kontrolle über die Lebensdauer des Objekts übernehmen und damit die Arbeit mit dynamisch erzeugten Objekten extrem vereinfachen.

Die Besonderheit des shared_ptr gegenüber anderen Smart Pointer (z.B. auto_ptr) ist, dass mehrere Instanzen des shared_ptr gleichzeigtig auf ein einzelnes Objekt verwiesen können.

Ein shared_ptr kann den Besitz eines Objektes teilen, auch wenn ein anderes Objekt in ihm gespeichert ist. Das kann dazu benutzt werden, um Zeiger auf Member-Objekte zu speichern, während der shared_ptr das Objekt besitzt, zu dem das Member-Objekt gehört.

Um die Lebensdauer des Objekts zu kontrollieren verwendet er einen Referenzzähler. Jede Kopie inkrementiert diesen Zähler, jeder Destruktor dekremiert ihn. Ist der Zähler bei Null angekommen, wird das Objekt gelöscht.

==========

Polling bezeichnet in der Informatik die Methode, den Status eines Geräts aus Hard- oder Software oder das Ereignis einer Wertänderung mittels zyklischem Abfragen zu ermitteln.

Здаров Серега!

Извини за запоздалый ответ.

Правильно сделал, что в личку написал, не дело в коментах дискуссию разводить.

В принципе здесь все просто. Я тебя с одной стороны хорошо понимаю, если бы на мою страну столько лили я бы тоже иголки навострил. К сожелению енто только рефлексы а не обдуманные мысли.

Прав ты в том, что многие русские (а может и большинство) попались на удочку пропаганды и видят на Украине только фашизм и тд. и тб.

Но вот меня туда причислять никак не стоит, я русские новости вообще не смотрю и сам к евлению „зато Крым наш“ очень негативно отношусь. Я совсем не против что он сейчас в ходит в состав России, но против многого другого связанного с ентим.

Как не странно ты меня еще десять лет назад вполне понимал. Помню как-то мы затронули ентот вопрос когда еще в „ТГ“ учились и на перерыве к тебе зашли. И ты признал, что да, есть такой косяк.

Во вторых говорить что в России все оболваненые а на Украине глаголят только истину, енто смех... Я однажды попал на украинское ТВ и продержался почти час. Енто не только сопостовимо с русским но даже хлеще.

А то что твариться на Украине с историей (если ее так назвать мона) так енто вообще караул и об этом мне кстати всегда мои друзья украинцы ссылки скидывали.

Я кстати ни разу не слышал в России „хохлов на вилы“, или тому подобное. Да и вообще в России всегда к украинцам лучше относились, чем к русским на Украине, надеюсь после всей ентой заворушки русские оклимаются станут воспринимать украинцев так же как и всегда – как братский народ.

Вот так и получается, что стравили два братских народа и каждый видет правду на своей стороне, а она как всегда где-то посиредине.

К сожалению я от тебя уже много глупостей слышал, вот только не отвечал на них

Твои слова: типо нас на Украине дохрена умных людей найдем правителя… Глупость не в умных людях, а про выборы. Твое мнение о „высадке“ немецкой армии на берегах Англии. Так же как и ссылка которой ты делился, где бедный донбас надо кормить. Ну и многое другое.

Так же и то, что Россия мочит Украину. Вопервых, кто такая Россия, я такую женщину не знаю, ну а во вторых, если бу „она“ захотела, то Украины уже не было бы и я совсем не о военном воздействии. А точней ее уже как тристо лет даже как княжества не було бы. И вообще не благодарность многих украинцев все нормальные нормы переваливает. Мало того что украина из гетманского поля (никем непризнанного!) засчет подарков “России“ (80% территории Украины!!!) в самую большую страну Европы превратилась, так мне здесь еще и пытались некоторые втереть, что украинскую армию русские развалили.

Правильно то, что правительство России пытается оставить за собой рычаги давления на Украину. Да, это не совсем красиво и я этого не поддерживаю, но такова мировая политика.

Ну а вообще здесь ничего не обяснишь в письменном виде. Получалось так, что при встрече я с моими друзьями из Украины часто на эту тему говорил и мы всегда приходили к примерно одному и тому же. Видели все одинаково, только по состоянию различной нацианальности ощущали немножко по разному, но это нормально.

Короче я отношусь абсолютно нормально к Украине как к (абсолютно) независимому государству и пусть хоть в Европе, хоть в Африке… хочется просто хороших отношений и взаимопонимания. Я например перестал общаться с некоторыми людми и западной

Украины которые с утра до вечера грязь на всю Россию и людей лили, и даже на мою стенку енто дерьмо кидали, кого ху… они меня в друзья приглашали я так и не понял. Ну а с другими общаюсь как и прежде, даже если мнения расходятся.
И вообще уже очень давно хочу в Киев…

Вот теперь и подумай кому мозги успели промыть, а кто может конечно и немного предвзято, но все же без ненависти и объективно о всю эту ситуацию рассматривает. Я в отличии от тебя не закрываю глаза на проблемы моей страны. С началом украинского конфликта у меня не изменилось не мнение о Путине, не о всей российской власти, ни даже о украинцах. Но Сережа, когда ты говоришь, что на Украине нет фашистов, нет ненависти к Русским, что происшествие в Одессе для тебя второстепенны а вот „небесная сотня“ это святые мученики за благое дело, то извини - мне с тобой говорить не о чем.

И да, я к войне в Чечне всегда очень негативно относился, и ты прав, там можно легко найти параллели между этими двумя конфликтами.
Вот и подумая, я считаю, что с Дудаевым, который намного дальше по духу и национальной пренадлежности от России, чем любой украинский сепаратист от Украины, можно было договориться весьма мирным способом – мужик не идиот был. Так может все-таки стоило договориться и жить в одной стране, но по федеральной системе?! Как живут столетиями и швейцарцы и канадцы, … без существиных проблем, чем первым делом провоцировать восток уже и так воспринявший майдан в штыки.
Ты помнишь, куда Россия окунулась когда попытались нагнуть чечьню силой?
Вот и подумай нужно ли это Украине?! Хоть там и не экстремальные исламисты, но хорошего от этого мало будет.