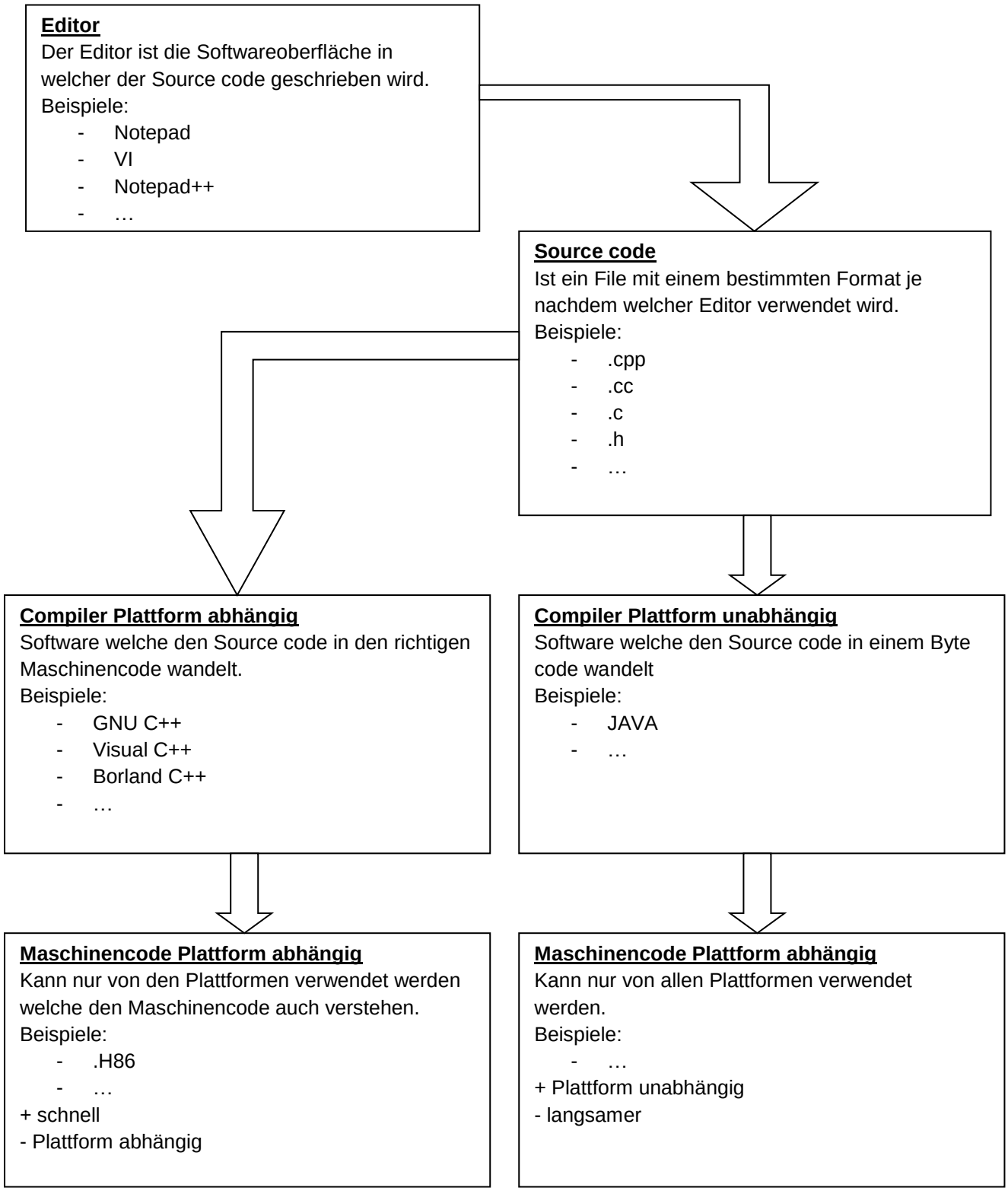
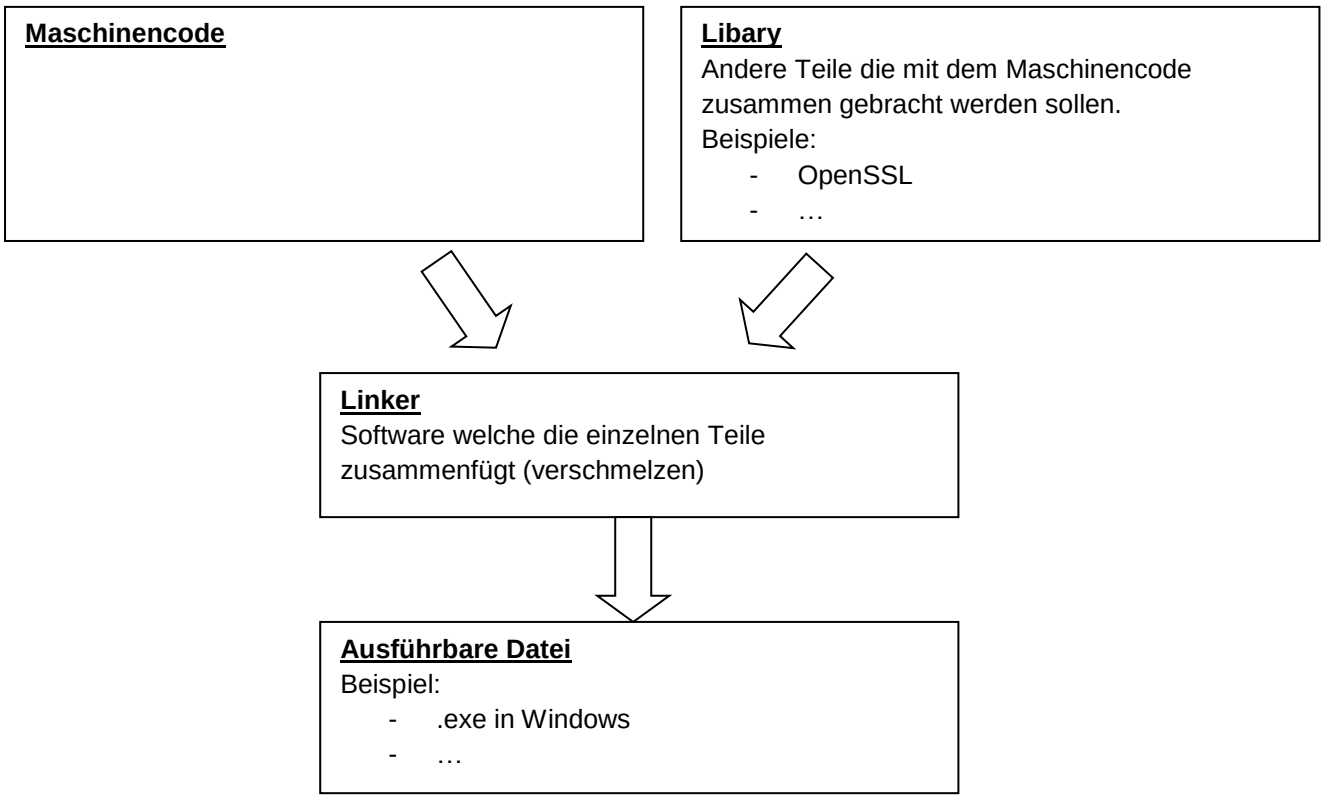
Strukturiertes Programmieren C++

Link: http://de.cppreference.com/w/
Link: http://www.cplusplus.com/

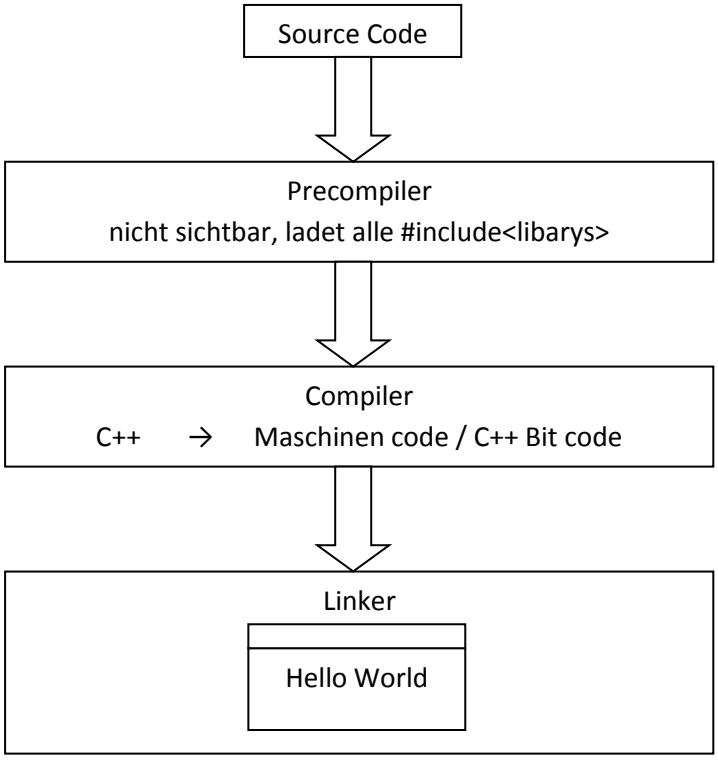
INHALTSVERZEICHNIS

1. Ablauf beim Software erstellen





2. Weg vom Source-code bis zum Execute Datei



3. Speicher reservieren

Damit ein Speicherbereich im System zugeordnet werden kann muss man folgende Schritte mache:

1. einen Datentypen bestimmen

   Es stehen folgende Grunddatentypen zur Verfügung:

   - *char* *einzelnes ASCII Zeichen*
   - *int* *ganze Zahlen*
   - *long* *ganze Zahlen*
   - *short* *ganze Zahlen*
   - *float* *reelle Zahlen*
   - *double* *reelle Zahlen*
   - *bool* *Wahrheitstabelle 1 oder 0*

Ausführbare Datei
Beispiel:
- .exe in Windows
- …

2. den Datentypen deklarieren

   Im Header des Programmes / Code müssen die Datentypen deklariert werden, zum Beispiel für die Zahl x muss eine Ganzzahl deklariert werden:

int x;	*Variable x deklarieren*
int x , y, z;	*mehrere Variablen des gleichen Typen in einer Zeile deklarieren*
int x (1);	*Variable x deklarieren, Wert in Klammer der Variable x zuweisen*
int x (1), y (2), z (3);	*mehrere Variablen des gleichen Typen in einer Zeile deklarieren mit jeweils einem Wert zuweisen*

   Achtung:
   - immer mit ; abschliessen = Zeile mit Code fertig!
   - wenn keine Wert zugewiesen wird kann irgendwas im Speicher sein, also immer zuerst „nullen“ mit (0)

Der Speicherplatz wird sobald das Programme / Code ausgeführt wird durch das Betriebssystem zur Verfügung gestellt. Dieser Platz kann irgendwo sein, das Betriebssystem sucht freien Platz und reserviert diesen dann für das Programm / Code.

4. Arbeiten mit MinGW mit Compiler in Eingabeaufforderung (CMD)

Folgende Befehle sind in der Aufgabeaufforderung (MS-Dos) für den Notepad++ zu gebrauchen:

Link zu cmd: C:_D_\KTSI_140215\03_PG_Programmieren\XX_Software\MinGW\console.bat

Befehl	**Auswirkung**
notepad++	Starten des Editor Software zum programmieren

cd uebungen	In das Verzeichnis Uebungen einsteigen
cd..	Aus dem jetzigen Verzeichnis herausspringen
dir	Auflisten aller Files im momentanen Verzeichnis
g++ nasa.cpp	Editorfile nasa.cpp im momentanen Verzeichnis kompilieren und als a.exe speichern
g++ nasa.cpp –o nasa.exe	Editorfile nasa.cpp im momentanen Verzeichnis kompilieren und als nasa.exe speichern
g++ nasa.cpp –c	Editorfile nasa.cpp als Byte code File nasa.o abspeichern

5. Arbeiten mit Dev-C++ mit Compiler in Eingabeaufforderung (CMD)

Link zu Software: C:\Users\chdmmn\Dev-Cpp\devcpp.exe

Minimum Code:

```
1. #include<iostream>
2. using namespace std;
3. in main ()
4. {
…
…
n. system („pause“);  //Alternativ kann auch (““) gemacht werden, dann hat man einen
                      automatisches schliessen vom CMD Fenster
n. }
```

6. Dezimal Operationen

Regeln: Wenn in einer Operation einmal eine Integerzahl vorkommt, so ist auch das Resultat nur noch eine Integer Zahl!

Beispiel: double = float * int = int (Ganzzahl!)

7. Bitweise Operationen

Regeln: **AND vor OR!**
Addition vor Subtraktion

NICHT [!] 4bit breit: !1010 = 0101
NICHT [!] 8bit breit: !1010 = 11110101

Zeichencompiler : && = ^

| |	=	v
!	=	-

ACHTUNG: Dualoperanden bedeutet, dass eine Zahl Dez / Okt / Hex zuerst in eine Dual umgewandelt werden muss bevor die Operation ausgeführt werden kann (Rechenleistung)

8. Vergleichsoperationen

==	gleich
!=	nicht gleich
<>	ungleich
<	kleiner als
>	grösser als

9. Befehle und Erklärungen

#include<……..>	**Bibliothek einfügen** Mit diesem Befehl wird eine Bibliothek eingefügt. In dieser sind Befehle und Funktionen definiert. So kann zum Beispiel nicht mit Wurzeln gerechnet werden wenn nicht die Bibliothek cmath im Programm eingefügt wird. Die wichtigsten Bibliotheken sind: iostram — Input / Output Grundfunktionen cmath — Mathematische Grundfunktionen Die Bibliotheken müssen immer auf der obersten Linie im Code stehen! Zeile 1: *#include<iostream>* Zeile 2: *#include<cmath>* Zeile 3: …….. Bibliotheken — Link Netz: http://www.cplusplus.com/reference/ iostream — Alles was mit Eingabe und Ausgabe zu tun hat und jede Menge Grundfunktionen. Link Netz: http://www.cplusplus.com/reference/iostream/ cmath — Mathematische Operatoren wie z.Bsp Wurzel, usw. Link Netz: http://www.cplusplus.com/reference/cmath/?kw=cmath string — Alles um den String, dass meiste sollte aber in iostream bereits integriert sein. Link Netz: http://www.cplusplus.com/reference/string/string/?kw=string http://www.cplusplus.com/reference/string/ http://en.cppreference.com/w/cpp/string/basic_string ctime — Alles um die Zeit des Systemes. Link Netz: http://www.cplusplus.com/reference/ctime/ vector — Alles zu vectoren. Link Netz: http://www.cplusplus.com/reference/vector/vector/?kw=vector

	algorithm Alles zu Algorithmen (suchen, finden, sortieren, Datenablage, usw.). Link Netz: http://www.cplusplus.com/reference/algorithm/?kw=algorithm
using namespace std;	**Namenbereich Standard definieren** Im Gültigkeitsbereich des Files wird der Namensbereich (**namespace**) mit dem Namen **std** zur Verfügung (**using**) gestellt. Man kann das auch weglassen, dann muss man aber z.B. **cout** durch **std::cout** ersetzen, weil die globale Variable **cout** zum Namensbereich **std** gehört. Alle Klassen/Funktionen aus dem C++ Standard existieren im Namensbereich **std,** deshalb **using namespace std.**
……..; *//*	**Informationstext am Ende einer Code Zeile** Alles nach den beiden Querstrichen (Slash) wird vom Programm nicht als Code angeschaut. Es besteht somit die Möglichkeit hier Informationen ausserhalb des Codes einzufügen. Beispiel: *cout<<"Hallo";* *// es wird Hallo ausgeworfen*
/*….. **…….*/**	**Informationstext als Block über mehrere Zeilen** Alles zwischen den Sternen wird vom Programm nicht als Code angeschaut. Es besteht somit die Möglichkeit hier Informationen ausserhalb des Codes einzufügen. Beispiel: */* Michael Mislin; 28.02.2014:* *Das folgende Programm wird den Text Hallo auswerfen* */* *cout<<"Hallo";*
{ ..Sequenz.. }	**Sequenz** Alles in der geschweiften Klammer ist eine Sequenz. Es können verschieden Sequenzen verschachtelt sein.
;	**Zeilenende Code** Der Strichpunkt (Semicolon) muss an jedem Zeilenende vom Code als Ende angegeben werden, ansonsten wird ein Fehler ausgegeben.
endl;	**Zeilenwechsel** Bewirkt in Programm einen Zeilenwechsel auf die nächste Zeile.
Variablen deklarieren	**Variablen deklarieren** Um mit Variablen Arbeiten zu können müssen diese deklariert werden. Der Schritt „Deklaration“ besteht aus dem Teil Datentypen, der Variablen und dem Wert. Datentypen: *char* *einzelnes ASCII Zeichen* *int* *ganze Zahlen* *long* *ganze Zahlen* *short* *ganze Zahlen* *float* *reelle Zahlen* *double* *reelle Zahlen*

	bool	*Wahrheitstabelle 1 oder 0*
	string	*Zeichenkette*
	Varaibel:	
	x, y, X, Y	Achtung: X <> x, es handelt sich um zwei unterschiedliche Variablen!
	Wert:	
	(3.14..)	*Es kann einer Variablen oder aber einer Konstanten (Bsp.: PI=3.14..) ein Grundwert gegeben werden.*
	int x;	*Variable x deklarieren*
	int x , y, z;	*mehrere Variablen des gleichen Typen in einer Zeile deklarieren*
	int x (1);	*Variable x deklarieren, Wert in Klammer der Variable x zuweisen*
	int x (1), y (2), z (3);	*mehrere Variablen des gleichen Typen in einer Zeile deklarieren mit jeweils einem Wert zuweisen*
	const int x = 1;	*durch das const wird die Variable zur einer Kontanten, das bedeudet der Wert kann n icht verändert werden!*
	char c;	*Char eröffnen aber nicht vorbelegen!*
	char c = 'W';	*Char mit einem Zeichen W vorbelegen*
	char c = ' ';	*Char mit einem Leerschlag vorbelegen ACHTUNG: nur 1 Zeichen hat Platz. Wenn mehrere zwischen ' ' dann wird nur das erste Zeichen darin gespeichert. Wenn ' ' dann MUSS ein Zeichen darin sein!*
	char a[13];	*char-Array mit 13 Zeichen reserviert aber leer*
	char a[ ] = "Hallo Welt!";	*char-Array mit 11 Zeichen vorbelegt mit dem Text „Hallo Welt!“.*
	ASCII Wert ermitteln	
	char c='a';	Deklarieren c und mit Zeichen a initialisieren
	int x=c;	Deklarieren x und mir ASCII Wert von a initialisieren
	cout<<x ;	c enthält den ASCII Wert von Buchstaben a
	#include <cstring>	*WICHTIG: immer Bibliothek STRING integrieren*
	string z = "Hallo Welt!";	*Erstellen eines String mit Inhalt „Hallo Welt!“*
	Erstellen & Auswerfen String	
	string z.kette;	*Erstellen einer Variable String,..*
	z.kette = "Hallo";	*..speichern Zeichen „Hallo“ in der Variabel..*
	cout << z.kette << endl;	*..auswerfen der Variable z.kette*

	Einlesen & Ausgeben String: *string zeichenkette;* — *Erstellen einer Variable String,..* *cin >> zeichenkette;* — *..einlesen.* *cout << zeichenkette;* — *..ausgeben der Eingabe.* *ACHTUNG: bei cin wird das Eeinlesen beim ersten Leerschlag abgebrochen!* *Beispiel Eingabe: Hallo Welt -> nur Hallo wird in Variable „Zeichenkette“ gespeichert, der Rest „ Welt“ geht verloren.* *Wenn String Texte mit leerschlag eigelesen werden sollen, so muss mit getline(cin,zeichenkette) eingelesen werden!* *Einlesen string ohne Bedingung* *string zeichenkette;* — *Erstellen einer Variable String,..* *getline(cin, zeichenkette);* — *..einlesen aber max. bis zum Ende einer Zeile (\n)..* *cout << zeichenkette;* — *..ausgeben der Eingabe.* *Einlesen string mit Bedingung:* *string zeichenkette;* — *Erstellen einer Variable String,..* *getline(cin, zeichenkette ,’y’);* — *..einlesen aber max. bis ein y eingegeben wird..* *cout << zeichenkette;* — *..ausgeben der Eingabe.* *Zusammensetz von strings:* *string string1, string2, string3;* — *Erstellen von drei Variablen String,..* *string1 = "Hallo ";* — *Zuweisen string1 Text “Hallo ”* *string2 = "Welt";* — *Zuweisen string2 Text “Welt”* *string3 = string1 + string2;* — *Zuweisen string3 Text aus string 1 & -2* *cout << string3 << endl;* — *Auswerfen string3* *string3 += " - " + string1 + "X";* — *Zuweisen string3 Text “ - Hallo X“* *cout << string3 << std::endl;* — *Auswerfen string3* Länge eines String bestimmen: *int* i = string_x.*lenght();* — i enthält die Anzahl Zeichen der Zeichenkette *int* i = string_x.*size();* — i enthält die Anzahl Zeichen der Zeichenkette String kopieren: *string* source = "Hello" ; — Deklarieren string „source“ und init. mit „Hallo“ *string* target(source); — Deklarieren string „target“ und Inhalt von source in target kopieren Zugriff auf das erste Zeichen eines String: *char* ch = string_x [0] ; — ohne Indexprüfung *char* ch = string_x.at (0) ; — mit Indexprüfung
Stoppuhr	*#include <ctime>* — *WICHTIG: immer Bibliothek integrieren* *Durch die Bibliothek ist eine neuer Datentyp clock_t verfügbar, für den genau gleich wie z.Bsp int eine Variable deklariert werden kann.* *clock_t start, stop;* — *Deklarieren Variablen* *start = clock();* — *Stoppuhr starten*

	..(zu messender Code).. *stop = clock();* *Stoppuhr stoppen* *cout* << ((*double*) *stop - start*) / *CLOCKS_PER_SEC* << *"s" ;* *Zeit rechnen & auswerfen*
Pausenzeit	*#include <windows.h>* *Sleep(200);* Die Zeiteinheit der 200 ist millisekunden [ms] !
rand () **&** **srand**	*Zufallszahlen generieren mit Funktion rand () & srand* *rand = einmal compiliert sind es immer dieselben Zahlen* *srand = initialisieren rand mit Systemzeit und somit gibt es immer andere Zahlen* *WICHTIG:* *#include <cstdlib>* *immer Bibliothek integrieren um rand zu benutzen* *#include <ctime>* *und um srand zu initialisieren* *srand((unsigned) time(0));* *Initialisieren von rand Funktion mit Systemuhr mit dem Wert Sekunden damit nicht immer die gleiche Zahl kommt wenn das Programm neu gestartet wird. Dies muss pro Programm einmal gemacht werden, am besten immer gleich nach Einstieg in das main. Somit wird jedes mal beim Starten des Programmes rand mit einer neuen Sekundenzahl initialisiert und es kommen bei jedem Programmdurchlauf andere Zahlen.* *int x = rand ();* *Zufallszahl von 0 und rand maximum erstellen* *int x = rand () % n;* *Zufallszahlen von 0 bis n-1 erstellen* *float x= 1 / (float) rand ();* *Zufallszahlen Gleitkomma von 0 bis 1* *int x = rand () % (max - min + 1) + min;* *Zufallszahlen zwischen min (inkl. min) bis max (inkl. max) max erstellen*
Casting	Es gibt zwei Arten von Casting 1. Von einer Gleitpunktzahl in eine Ganzzahl wandeln: Automatisches casten int a = 9; b = 2; c (0); c = a / b c =4 (0.5 wird abgeschnitten) Manueles casten double a = 1.123; int b (0); b = (int) a a wird als Wert 1 in b abgespeichert, Rest wird abgeschnitten b = 1 2. von Ganzzahl in eine Gleitpunktzahl wandeln: double a = (0);

	int b (1); a = (int) b b wird als Wert 1.0 in a abgespeichert, Rest geht verloren a = 1.0 ACHTUNG: Casting schneidet nur ab, es wird nie gerundet! *CASTING:* *float x (1.123);* *es wird eine Variable Float erstellt mit dem Wert 1.123.* *x = (int) x;* *der Wert Float wird mittels CASTING in einen Wert Integer umgewandelt. Dazu wir alles was hinter dem Punkt steht abgeschnitten.* *cout<<x;* *der Wert von x beträgt jetzt 1 weil Integer eine Ganzzahl ist.*
cout<<…;	**Auswerfen / Anzeigen** Der Teil nach << Zeichen wird ausgeworfen. Es kann sich dabei um eine deklarierte Variable handeln oder aber um einen ASCII Text. Falls es sich um einen Text handelt so muss dieser zwischen zwei "…" befinden. Wenn nach dem Text noch ein weiterer Befehl wie Zeilenwechsel kommt so muss zwingen am Ende noch << gemacht werden. Beispiel: *cout<<"Hallo";* Auswurf **Hallo** ohne Zeilenwechsel *cout<<"Hallo"<<endl;* Auswurf **Hallo** mit Zeilenwechsel *cout<<x;* Auswurf **Wert** der Variable **x** ohne Zeilenwechsel *cout<<x<<endl;* Auswurf **Wert** der Variable **x** mit Zeilenwechsel *cout<<"Ergebnis: „<<x<<endl;* Auswurf Text **Ergebnis** + **Wert x** + Zeilenwechsel im Programm
Steuerzeichen Escabe Sequenzen	Innerhalb eines Textes (zwischen "..." kann mittels verschiedenen Steuerzeichen vereinfacht verschiedene Aktionen ausgeführt werden. Siehe auch Link: http://de.cppreference.com/w/cpp/language/escape *cout<<"**\a**Hallo";* **\a** steht für BEL (bell) gibt ein akustisches Warnsignal *cout<<"**\b**Hallo";* **\b** steht für BS (backspace) der Cursor geht eine Position zurück *cout<<"**\f**Hallo";* **\f** steht für FF (formfeed) ein Seitenvorschub wird ausgelöst *cout<<"**\n**Hallo";* **\n** steht für NL (new line) der Cursor geht zum Anfang der nächsten Zeile *cout<<"**\r**Hallo";* **\r** steht für CR (carriage return)

	der Cursor geht zum Anfang der aktuellen Zeile *cout<<"**\t**Hallo";* **\t** steht HT (horizontal tab) der Cursor geht zur nächsten vertikalen Tabulatorpostion in der aktuellen Zeile *cout<<"**\v**Hallo";* **\v** steht für VT (vertical tab) der Cursor geht geht zur nächsten vertikalen Tabulatorenposition *cout<<"**\>>**Hallo";* **>>** wird ausgegeben *cout<<"**\'**Hallo";* **'** wird ausgegeben *cout<<"**\?**Hallo";* **?** wird ausgegeben *cout<<"****Hallo";* **** wird ausgegeben usw.
cin>>…;	Einlesen / Erfassen Der Teil nach dem >> wird im Code abgefragt, sprich man muss was eingeben. Wenn „etwas“ eingelesen wird so muss das immer in eine Variable geschrieben werden. Beispiel: …. *int x;* Deklaration Variable Ganzzahl x ohne Wert *cin>>x;* Benutzer kann eine ganze Zahl eingeben und …. mit ENTER die Eingabe beenden. …. *char x;* Deklaration Variable ASCII-Zeichen x *cin>>x;* Benutzer kann ein ASCII-Zeicheneingeben und …. mit ENTER die Eingabe beenden.
sizeof(…)	Grösse ausgeben Syntax gibt die Grösse des Datentyp in Byte aus. a = sizeof (x); in a steht die Grösse in Byte der Variable x b = sizeof (double); in b steht die Grösse in Byte vom Datentyp double Beispiel: <u>Code</u> … *bool b=0;* *cout<<"Bool = "<<sizeof(b)<<" Byte"<<endl;* … <u>Ausgabe am Bildschirm</u> …. *Bool = 1 Byte* ….

	<u>Grösse Array ermitteln</u> Achtung: Funktioniert nicht bei dynamischen Arrays! Die Anzahl Felder eines Arrays kann anhand des Dividend von „Array Speicherplatz Gesamt“ (sizeof(Array)) und des „Speicherplatz ein Feld“ (sizeof(Array[0]) berechnet werden. *int main (){* *const int x (100);* *int testArray [x];* *int sizeArray (0);* *sizeArray = sizeof(testArray)/sizeof (testArray [0]);* *cout<<"\n\tGroesse Feld [0] Array: "<<sizeof testArray [0]; // Ausgabe = 4Byte* *cout<<"\n\tGroesse Gesamtes Array: "<<sizeof(testArray); // Ausgabe = 400Byte* *cout<<"\n\tAnzahl Felder[n] Array: "<<sizeArray; // Ausgabe = 100 Felder* *return(0);* *}* Info: Gesamt Byte durch Byte eines Feldes ergibt Anzahl Felder!
sqrt ….	Quadratwurzel berechnen Es wird die Quadratwurzel aus der folgenden Variable berechnet. Beispiel: *sqrt 4;* Wurzel aus 4 berechnen
pow (… ,…)	Potenzieren Es wird die Basis mit der Potenz potenziert. Beispiel: *pow (2,3)* 2 x 2 x 2 wird gerechnet Achtung: *#include<cmath>* wird benötigt!
%	Modulo Modulo gibt den Rest einer Division aus! Bsp: 10 / 3 =3, Rest 1 In diesem Beispiel ist 1 das Resultat der Berechnung Modulo. a = 10 % 3; Der Rest 1 wird in die Variable A geschrieben.
Logische Operatoren	|| Oder *(Taste **Alt Gr** & **7**)* & Und ! Nicht
if *if (a == b)* *{*	WENN Abfrage if (a == b) Bedingung

….Sequenz… *}*	Wenn die Bedingung a == b true ist, dann wird die Sequenz in der Klammer ausgeführt
else if *else if (a == b) {* *….* *…. Sequenz …* *}*	SONST WENN Abfrage Kann nur nach einer if Abfrage gemacht werden. Gleicher Syntax wie bei if
else *else* *{* *…. Sequenz …* *}*	SONST Default Keine Bedingung! Wird ist zuvor im Code das if und die n if else nicht true, dann wird der Code in der { Sequenz } nach dem else ausgeführt.
switch *switch (a)* *{* *case b: cout<<…;* *break;* *case c: cout<<…;* *break;* *default: cout<<…;* *}*	Schalter Abfrage ACHTUNG: nur mit Integer Werten anstelle von a, b, c anwendbar! Ist eine statisch Abfrage, heisst in dieser Anwendung kann man nicht nochmals eine weitere Verschachtelung machen. Wenn eine Bedingung zutrifft kann keine andere mehr zutreffen. Anwendungsbeispiel: Würfel! Dieser hat immer nur 6 Seiten und muss nie erweitert werden. **switch (a)** **Abfrage** Variable die Abgefragt werden soll, a = Bedingung { Sequenz starten **case b: cout..** **Fall 1** Im Fall, dass in b == a ist soll Code hinter Doppelpunkt ausgeführt werden **case c: cout..** **Fall 2** Im Fall, dass in c == a ist soll Code hinter Doppelpunkt ausgeführt werden **case n: cout..** **Fall ..** Es können unendlich viele case geamcht werden **break;** **Abbruch Sequenz** Muss zwingen gemacht werden damit der switch abgebrochen wird, sonst wird der code ausgeführt bis zum nächsten break! } Sequenz beenden
for *for (int i = 0; i<5 ; i++)*	Schleife Kopfgesteuert mit Zähler (Loop) ACHTUNG:

{

…. Sequenz …
}

Wenn nur eine Zeile Code nach der Schlaufe gebraucht werden, so muss keine geschweifte Klammer verwendet werden. So wird nur die erste Zeile als zur for Schlaufe ausgeführt.

int i=0 **Initialschritt**
heisst, es wird die Variable i erstellt und der Wert 0 reingeschrieben. Die Variable kann irgendeinen Name haben aber darf pro Programm nur einmal vorkommen wenn es mehrere Schlaufen gibt.
Initialschritt kann weggelassen werden wenn die zu prüfende Variable bereit besteht!

i < 5 **Bedingung**
Schlaufe wird ausgeführt solang die Bedingung erfüllt wird, in diesem Fall solange i ist kleiner als 5 ist.

i++ **Regel**
In diesem Fall eine Inkrementierend

{ Sequenz } **Schleifenrumpf**
In diesem Teil ist der auszuführende Code

Beispiel:

Ablauf:	**Schritt**	**i Wert**
1. Initialschritt	int i = 0	0
2. Prüfen auf true	i < 2 *(ist true)*	0
3. Schleifenrumpf ausführen	{ Sequenz }	0
4. Regel ausführen	i++	1
5. Prüfen auf true	i < 2 *(ist true)*	1
6. Schleifenrumpf ausführen	{ Sequenz }	1
7. Regel ausführen	i++	2
8. Prüfen auf true	i < 2*(ist false)*	2
9. Schleife abbrechen weil false!		

In diesem Beispiel wird die Schlaufe also 5 x Durchlaufen!

Wichtig:
Wenn Bedingung == true wird Schlaufe ausgeführt, wenn Bedingung == false wird die Schleife abgebrochen.
Es kann auch ein Zeichen, also eine char oder string Variable in die Bedingung eingebracht werden.

while

while (a < 5 && b < 3)
{

…. Sequenz …
}

Schleife Kopfgesteuert (Loop)

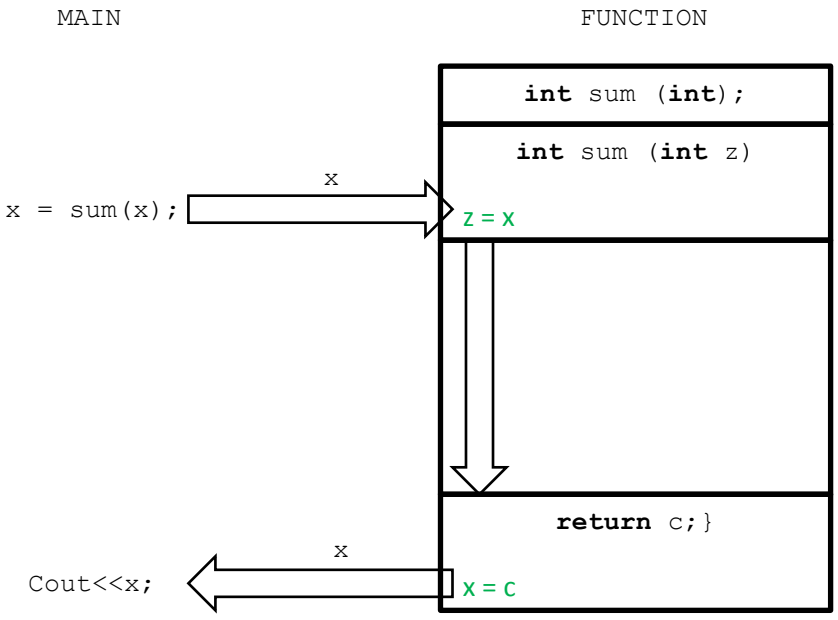
Kopfgesteuert weil die Bedingung / Prüfung am Anfang geprüft wird!

(a < 5 && b < 3) **Bedingung**
Hier lautet die Bedinung a < 5 UND b < 3. Solange dies erfüllt ist wird die Schlaufe ausgeführt. Bedingung muss true geben sonst wird Schlaufe nicht ausgeführt.

Sequenz } **Schleifenrumpf**
In diesem Teil ist der auszuführende Code

do while

Schleife Fussgesteuert (Loop)

do { *…* *… Sequenz …* *…* *} while (again == 'Y');*	Fussgesteuert weil die Bedingung / Prüfung am Ende der Schleife ist! **(again == 'Y');** **Bedingung** In diesem Fall ist die Bedingung ein CHAR welche Y enthalten muss. Solange in diesem CHAR kein Y steht wird die Schlaufe ausgeführt. Diese Art von Schlaufe eignet sich darum bestens für eine Abfrage: Für eine neue Berechnung jetzt Y drücken! Bedingung muss true geben sonst wird Schlaufe nicht ausgeführt. **Sequenz }** **Schleifenrumpf** In diesem Teil ist der auszuführende Code Beispiel: *do {* *c = a + b;* *cout<<"Neue Berechnung? Wenn Ja bitte jetzt y drücken : ";* *cin>>again;* *} while (again == 'Y' || again == 'y');* Am Ende der Schlaufe wird CHAR again mit einem ASCII Zeichen beschrieben, wenn diese Y oder y ist, dann wird die Schlaufe wiederholt.
Funktionen	Beim Aufruf von Funktionen gibt es verschieden Arten und entsprechend werden diese aufgebaut und aufgerufen. **WICHTIG: Eine Funktion kann IMMER nur max. EINE Variable zurückgegeben!** **WICHTIG: Eine Funktion sollte immer einen sinnvollen Namen haben!** **WICHTIG: Werte werden von links nach rechts eine nach der anderen parallel übergeben!** Übersicht wie eine Funktion abläuft: 

1. Funktion MIT einer Übergabe von Variable(-n) und Rückgabe Variable

```
        #include <iostream>
        using namespace std;
        int add_function (int, int);            // Deklaration Funktion
        ….
2,3     int add_function (int x, int y); {      // Funktion Definition, Abarbeitung wenn aufgerufen
4,5     y = x + y ;
6,7     return y;  }
        ____________________________________________________________
        Main {
        int a (3), b (7), c (0);
        …
1       c = add_function (int b, int a );               // Aufruf der Funktion add_function
8       …
        }
```

Schrittablauf bei der Code-Zeile ***c = add_function (int a, int b);***

Schritt	a	b	c	x	y	Bemerkung:
1	3	7	0	en	en	Funktion wird aufgerufen und gestartet
2	3	7	0	?	?	Variablen x und y werden erstellt
3	3	7	0	7	3	Variablen x und y werden initialisiert mit Wert a und b
4	3	7	0	7	3	Code ausführen: 3 + 7 = 10
5	3	7	0	7	10	Code ausführen: y = 10
6	3	7	10	7	10	Rückgabe Wert an Variable c
7	3	7	10	en	en	Funktion fertig, Variablen sterben
8	3	7	10	en	en	Code weiter ausführen in Main Sequenz

en = existieren nicht
? = irgend ein undefinierter Wert, je nachdem was im Speicherbereich steht der reserviert wurde

2. Funktion MIT einer Übergabe von Variable(-n) und OHNE Rückgabe

OHNE Rückgabe = VOID

```
        #include <iostream>
        using namespace std;
        void add_function (int, int);           // Deklaration Funktion
        ….
2,3     void add_function (int x, int y); {     // Funktion Definition, Abarbeitung wenn aufgerufen
4,5     y = x + y ;
6,7     cout<<y ;  }
        ____________________________________________________________
        Main {
        int a (3), b (7), c (0);
        …
1       add_function (int b, int a );           // Aufruf der Funktion add_function
8       …
        }
```

Schrittablauf bei der Code-Zeile ***add_function (int b, int a);***

Schritt	a	b	c	x	y	Bemerkung:
1	3	7	0	en	en	Funktion wird aufgerufen und gestartet
2	3	7	0	?	?	Variablen x und y werden erstellt
3	3	7	0	7	3	Variablen x und y werden initialisiert mit Wert a und b
4	3	7	0	7	3	Code ausführen: 3 + 7 = 10
5	3	7	0	7	10	Code ausführen: y = 10
6	3	7	0	7	10	10 Ausgabe aus der Funktion heraus
7	3	7	0	en	en	Funktion fertig, Variablen sterben
8	3	7	0	en	en	Code weiter ausführen in Main Sequenz

en = existieren nicht
? = irgend ein undefinierter Wert, je nachdem was im Speicherbereich steht der reserviert wurde

3. Funktion OHNE Übergabe und OHNE Rückgabe Variable

OHNE Rückgabe = VOID

```
        #include <iostream>
        using namespace std;
        void add_function ();                  // Deklaration Funktion
        ….
        void add_function ();  {               // Funktion Definition, Abarbeitung wenn aufgerufen
2       int x (5), y (4) ;
3,4     y = x + y ;
5,6     cout<<y<<" ist das Resultat";  }
        ____________________________________________________________
        Main {
        int a (3), b (7), c (0);
        …
1       add_function ();                       // Aufruf der Funktion add_function
7       …
        }
```

*Schrittablauf bei der Code-Zeile **add_function ();***

Schritt	a	b	c	x	y	Bemerkung:
1	3	7	0	en	en	Funktion wird aufgerufen und gestartet
2	3	7	0	5	4	Variablen x und y werden lokal erstellt und initialisiert
3	3	7	0	5	4	Code ausführen: 5 + 4 = 9
4	3	7	0	5	9	Wert zuweisen: 9 = y
5	3	7	0	5	9	"9 ist das Resultat" Ausgabe aus der Funktion heraus
6	3	7	9	en	en	Funktion fertig, Variablen sterben
7	3	7	9	en	en	Code weiter ausführen in Main Sequenz

en = existieren nicht
? = irgend ein undefinierter Wert, je nachdem was im Speicherbereich steht der reserviert wurde

4. Funktion OHNE Übergabe und MIT Rückgabe Variable

```
        #include <iostream>
        using namespace std;
        int add_function (int);                // Deklaration Funktion
        ….
        int add_function ();  {                // Funktion Definition, Abarbeitung wenn aufgerufen
2       int x (5), y (4) ;
3,4     y = x + y ;
5,6     return y;  }
        ____________________________________________________________
        Main {
        int a (3), b (7), c (0);
        …
1       c = add_function ();                   // Aufruf der Funktion add_function
7       …
        }
```

*Schrittablauf bei der Code-Zeile **c = add_function ();***

Schritt	a	b	c	x	y	Bemerkung:
1	3	7	0	en	en	Funktion wird aufgerufen und gestartet
2	3	7	0	5	4	Variablen x und y werden lokal erstellt und initialisiert
3	3	7	0	5	4	Code ausführen: 5 + 4 = 9
4	3	7	0	5	9	Wert zuweisen: 9 = y
5	3	7	9	5	9	Rückgabe Wert an Variable c
6	3	7	9	en	en	Funktion fertig, Variablen sterben
7	3	7	9	en	en	Code weiter ausführen in Main Sequenz

en = existieren nicht
? = irgend ein undefinierter Wert, je nachdem was im Speicherbereich steht der reserviert wurde

Funktionen mit static

void testfunktion(){
int i = 0;

static → variable in der Funktion aber Speicherplatz wird nicht freigegeben wenn die Funktion verlassen wird (Umgehen der Sichtbarkeitsregel)

static int y = 0;

```
    static int y = 0;
    y++;
}
```

beim ersten eintritt wird 0 zugewiesen, nach dem zweiten eintritt wird der Code wie ausgelassen. Dies ist die einzige Möglichkeit die Sichtbarkeitsregel, also das freigeben Speicherplatz nach verlassen Funktion, zu umgehen.

Möglich für Zähler einer Funktion zu machen

Sichtbarkeitsregel:
Variable die unten im Code deklariert wird kann nicht weiter oben im Code aufgerufen werden!
Eine deklarierte Variable in einer Funktion ist nur innerhalb der Funktion sichtbar AUSSER wenn diese als Static deklariert wird.

Beispiel Funktion mit Schlaufendurchlaufzähler:

```
                        #include <iostream>
                        using namespace std;
                        void testfunktion();      // Deklaration Funktion

4,12                    void testfunktion(){
5,13,                   int i2 = 0;               // deklarieren und initialisieren
6,14,                   static int y = 0;         // Statisch deklariert, heisst Variable stirb nicht!
7,15,                   y++;
8,9,16,                 }

                        ____________________________________________________

                        int main () {
                        int x (99);
                        …
1,2,10,11,17,18,19      for(int i1 = 0; i1< 2; i1++){
4,12,                   testfunktion();
20                      }
21                      …
```

Schrittablauf bei der Code-Zeile ***c = add_function (int a, int b);***

Schritt	x	i1	i2	y	Bemerkung:
1	99	en	en	en	Einstieg in die for Schlaufe
2	99	0	en	en	Deklarieren i1 und initialisieren mit = 0
3	99	0	en	en	Bedingung prüfen: i1 = 0 < 2 == true
4	99	0	en	en	In Sequenz der for Schleife springen und Funktion aufrufen
5	99	0	0	en	Deklarieren i2 & initialisieren mit = 0
6	99	0	0	0	Deklarieren y als static & initialisieren mit = 0
7	99	0	0	1	Inkrementieren der Variable y = 0 + 1
8	99	0	en	en	Funkt. fertig, Variable i2 & y stirbt, Speicherplatz y bleibt erhalten
9	99	0	en	en	Sprung in Sequenz Main in die Schleife for
10	99	1	en	en	Inkrementieren der Variable i1 = 0 + 1
11	99	1	en	en	Bedingung prüfen: i1 = 1 < 2 == true
12	99	1	en	en	In Sequenz der for Schleife springen und Funktion aufrufen
13	99	1	0	en	Deklarieren i2 & initialisieren mit = 0
14	99	1	0	1	Deklarieren y als static & initialisieren mit = 1 aus Speicherplatz
15	99	1	0	2	Inkrementieren der Variable y = 0 + 1
16	99	1	en	en	Funkt. fertig, Variable i2 & y stirbt, Speicherplatz y bleibt erhalten
17	99	1	en	en	Sprung in Sequenz Main in die Schleife for
18	99	2	en	en	Inkrementieren der Variable i1 = 1 + 1
19	99	2	en	en	Bedingung prüfen: i1 = 2 < 2 == false
20	99	en	en	en	Schleife for beenden, Variable i1 stirbt
21	99	en	en	en	Code weiter ausführen in Main Sequenz

en = existieren nicht

Funktion überladen	Überladen heisst, dass die gleiche Funktion in der Main Sequenz mehrfach mit verschiedenen Übergabe-Variablen aufgerufen wird. Damit dies funktioniert muss

	dieselbe Funktion aber auch vor der Main-Sequenz deklariert sein und auch Aufgerufen werden! Wichtig: die Übergabe in den ( ) muss übereinstimmen mit der Funktion ansonsten funktioniert es nicht. Beispiel *#include <iostream>* *using namespace std;* *Void add (int, int);* *// Deklaration Funktion* *Void add (double, double);* *// Deklaration Funktion* *Void add (int a, int b);* *// Funktion abarbeiten* *{ ….. }* *Void add (double c, double d);* *// Funktion abarbeiten* *{ ….. }* _____ *Main {* *…* *add (int a, int b);* *// Aufruf funktioniert* *add (double c, double d);* *// Aufruf funktioniert* *add (int a);* *// Aufruf funktioniert NICHT weil nicht in dieser Version deklariert & // aufgerufen* *add (int a, double c);* *// Aufruf funktioniert NICHT weil nicht in dieser Version deklariert & // aufgerufen* *}*
Funktion Default Parameter *Void add (int x, int y = 0) {* *…* *}*	Default Parameter heisst, dass die Funktion z. Bsp. mit zwei Variablen deklariert wird aber in der Main Sequenz nur eine Variable übergeben wird. Für die zweite Variabel wird diese mit einem Default-Wert während dem Abarbeiten deklariert und initialisiert. - Falls die Funktion ohne zweite Variable in der Main Sequenz aufgerufen wird, wird die Funktion den static Wert übernehmen - Falls die Funktion mit einer zweiten Variabel in der Main Sequenz aufgerufen wird, wird der Wert vom Ausruf übernommen Aufruf ohne zweite Variabel in Main Sequenz: *#include <iostream>* *using namespace std;* *Void add (int, int);* *// Deklaration Funktion* *Void add (int x, int y = 0);* *// b mit dem = 0 als „Default“ Variable* *{ … }* *// deklariert* _____ *Main {* *int a (22), b (77);* *add (int a);* *// Aufruf funktioniert* *}* Bei diesem Beispiel wir beim Einsprung in die Funktion der Wert a = 22 in x gespeichert und in y = 0 gespeichert! Aufruf mit zweiter Variabel in Main Sequenz: *Void add (int, int);* *// Deklaration Funktion* *Void add (int x, int y = 0);* *// b mit dem = 0 als „Default“ Variable* *{ … }* *// deklariert* *Main {*

	int a (22), b (77); *add (int a, int b);* *// Aufruf funktioniert* } Bei diesem Beispiel wir während dem Abarbeiten in der Funktion der Wert a = 22 in x gespeichert und b = 77 in y gespeichert! Dies geht ABER NUR jeweils wenn die Funktion mit demselben Datentypen aufgerufen wird wie er in der Funktion selbst deklariert ist. Bei folgenden Beispiel kann nicht für double überladen werden. *Void add (int, int);* *// Deklaration Funktion* *Void add (int x, int y = 0);* *// b mit dem = 0 als „Default“ Variable* *{ … }* *// deklariert Main {* *int a (22), b (77);* *add (int a, double b);* *// Aufruf funktioniert nicht* } Es kann auf diesem Weg Code gespart werden. WICHTIG: beim Default Parameter muss immer von rechts nach links gearbeitet werden weil beim Aufruf der Funktion im Main die erste Variable in die erste Variable der Funktion übergeben wird, die zweite in die zweite usw.! Immer ab der ersten Default Variabel müssen alle weiteren auch Default gesetzt werden! Falsch: *Void add (int x = 0, int y, int z = 0);* Richtig: *Void add (int x, int y = 0, int z = 0);*
Referenzen	Call by reference: Heisst die Adresse der Variable wird in einer Referenz Variabel gespeichert! Vorteil ist, dass der Wert ohne eine dereferenzierung mit dem Operator * aufgerufen werden kann, jedoch ist die Referenz konstant der gleichen Variabel zugewiesen und kann nicht mehr verändert werden. **WICHTIG:** Ist einmal eine Referenz erstellt so ist diese Konstant und kann nicht mehr geändert werden, deshalb muss diese beim Deklarieren auch gleich initialisiert werden! Die Datentypen der Referenz und der Variable von welcher die Adresse in die Referenz geschrieben werden soll müssen übereinstimmen! **Referenzen verwenden** *int a = 7;* // Deklarieren und initialisieren *int &r_a = a;* // Deklarieren der Referenz r_a und initialisieren mit der Adresse von der Variable a *cout<<a;* // Ausgabe Wert 7 *cout<<r_a;* // Ausgabe Wert 7, Referenzen müssen nicht dereferenziert werden!

Refernezen bei Funktionen verwenden:

*void func(int &, int , int *);* // Deklaration der Funktion und der Variablen der Funktion

*void func(int &n, int m, int *p)* // Aufrufen der Funktion und initialisieren der Variablen

Pointer

```
int a (7);
int *p1;
*p1 = &a;
```

Pointer enthalten die Adressen von Variablen. Jede Variable muss eine Adresse enthalten damit das Programm die Variable auch immer wieder findet.

INFO:

Call by value: Heisst der Wert der Variable wird übergeben!
Nachteil ist, dass auf diese Art kann NUR ein Wert aus der Funktion zurückgegeben werden!

Call by reference: Heisst die Adresse der Variable wird in einer Referenz Variabel gespeichert!
Vorteil ist, dass der Wert ohne dereferenzieren aufgerufen werden kann, jedoch ist die Referenz konstant der gleichen Variabel zugewiesen und kann nicht mehr verändert werden.

Call by adress: Heisst die Adresse der Variabel wird übergeben!
Vorteil ist, dass auf diese Art alle so viel gewünschte Variablen aus der Sequenz der Funktion gelesen oder geschrieben werden, wie der Code es verlangt

WICHTIG:

Es kann nur eine Adresse in einem Pointer gespeichert werden!

* Operator „Deklarieren“ UND Operator „Dereferenzieren“, je nachdem wie der Operator eingesetzte wird

& Operator „Adresse“, Compiler weiss dass die Adresse von der Variable ausgelesen werden muss

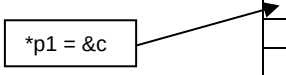
```
int *p = NULL;                  // Variable erstellen und mit NICHTS initialisieren.
cout<<*p;       0               // dereferenzieren, sprich Wert Variable lesen
                                // Compilerfehler oder „NICHTS“ weil der Wert nichts
                                // enthält

cout<<p;        0               // referenzieren, sprich die Adresse lesen
                                // ACHTUNG: da die Adresse mit „NICHTS“ gefüllt ist,
                                // zeigt das System dies als ein Zahl 0 an!
```

NULL macht Sinn wenn eine Variable beim Deklarieren gleich mit NULL (NICHTS) initialisiert wird. Wenn im Code konsequent immer ein nicht gebrauchter Pointer nach dem verwenden auf NULL initialisiert wird, kann z. Bsp. mittels der folgenden Prüfung ermittelt werden ob ein Pointer bereits von einem anderen Prozess besetzt ist oder ob er frei ist!

```
If ( p1 == NULL ) {….}
else if ( p2 == NULL ) {….}
else if ( pn == NULL ) {….}
…
```

Variable Adresse	*Variable mit Wert*	*Datentyp*



0Fx00000000	*Variable a, Wert 12*	*Integer*
0Fx00000001	*Variable b, Wert A*	*Char*
0Fx00000002	*Variable c, Wert 23.54*	*Double*
0Fx00000003	*Variable d, Wert "HALLO"*	*String*
0Fx……..	*Variable x, Wert n*	*x*
0Fx……..	*Variable x, Wert n*	*x*
0Fx……..	*Variable x, Wert n*	*x*
0Fx0000006D	*Variable e, Wert 54.3*	*Float*
0Fx0000006E	*Variable f, Wert 1*	*Bool*
0Fx0000006F	*Variable g, Wert "A"*	*Char*

int *p1 = &c; Variable P1 deklarieren und Adresse 0Fx00000002 von der Variable c im P1 abspeichern

cout<<*p1; 23.54 // dereferenzieren, sprich Wert Variable

cout<<p1; 0Fx00000002 // referenzieren, sprich die Adresse

cout<<&a 0Fx00000000 // Adresse von Variable a

WICHTIG:

p1 = a; FALSCH: Compiler Fehler! Wert von a kann NICHT im Pointer gespeichert werden

p1 = &a; RICHTIG: Adresse von der Variable a in p1 speichern

*p1 = a; FALSCH: Compiler Fehler! Deklarieren des Pointer p1 und initialisieren mit Wert von a, der Wert kann NICHT im Pointer gespeichert werden

*p1 = &a RICHTIG: Deklarieren Pointer p1 und initialisieren mit Wert von a

*p1 Deklarieren des Pointer p1, ohne initialisieren! Im Pointer ist irgend eine willkürliche Adresse abgespeichert

Beispiel Pointer, referenziert & dereferenziert

```
int main(){
1.)       int a(6), b(2);
          int *pa, *pb;

2.)       pa = &a;
3.)       *pa +=1;      // dereferenziert und deshalb a = a + 1
4.)       cout<<"a = "<<a<<endl;
5.)       pb = &b;
6.)       *pb=*pb**pa;   // dereferenziert und deshalb b = b * a
7, 8, 9.) b = b++ + ++a; // 7.) a + 1, dann 8.) b = b + a, dann 9.) b ++
10.)      cout<<"b = "<<b<<endl;
          return 0;}
```

Schritt	a	b	pa	pb	Ausgabe
1	6	2			
2			0Fx.. von a		
3	7				
4					a = 7
5				0Fx.. von b	
6		14			
7	8				
8		22			
9		23			
10					b = 23

Pointer conversion (Casting mit Pointer)

```
int main(){
        int a = 72, b =2 ;              // Deklarieren und initialisieren der Variablen

        int *p_i = &a ;                 // Deklarieren  Pointer p_i  und Adresse  von a in Pointer speichern
        char *p_c1  (NULL);             // Deklarieren Char Pointer p_c  und initialisieren mit NULL
        …
        p_c1 = (char*)(p_i);            // Pointer konversion! Casten von p_i als Char in p_c1 aber nur
                                           Adresse wird in p_c1 geschrieben
        char *p_c2 = (char*)(p_i);      // Erstellen Char Pointer p_c 2 und gleichzeitig Pointer konversion!
                                           Casten von p_i als Char in p_c1 aber nur Adresse wird in p_c2
                                           geschrieben
        *p_c = 'z';                     // Dereferenziert einen neues Zeichen z in p_c1 schreiben
        cout<<*p_i;                     // Dereferenziert , also Wert 72 von a ausgeben
         cout<<*p_c1i;                  // Dereferenziert , also Wert  vom ASCII Zeichen. Der Wert 72
                                           entspricht H also „H“ wird ausgegeben.
}
```

Void Pointer (indirektes casten mit Pointer)

```
int main(){
        int a = 1.234,
        void *p_v = &a;                 // Void Pointer deklarieren und Adresse von a in Pointer speichern

        cout<<(int*) p_v;               // Ausgabe Adresse von p_v der auf a zeigt als Integer

        cout<<(double*) p_v;            // Ausgabe Adresse von p_v der auf a zeigt als Double

        cout<<*(int*) p_v;              // Ausgabe dereferenziert, also Wert von p_v als Integer

        cout<<*(double*) p_v;           // Ausgabe dereferenziert, also Wert von p_v als Double
}
```

Syntax deklarieren: void ***p_v** = &a;
Syntax Aufruf: ***(*variablentyp***) *p_v*

& = Adresse von der nachfolgenden Variable referenzieren
* = dereferenziert, also Wert von Variable auf welcher der Pointer zeigt
variablentyp = als was soll die der Wert von der Ursprünglichen Variable ausgegeben werden. Wenn nicht dereferenziert wird, dann ist Ausgabe einfach eine Adresse
* = als Pointer deklarieren
p_v = Pointer auf eine Variabel

Pointeranalyse:

```
int main(){
1.      float *f1, f2, f3(1), *f4;
2.      f2 = 23;
3.      float &f5 = f2;
4.      f1 = &f5;
5.      *f1 = 5;
6.      f1 = NULL;
7.      cout<<"f1: "<<f1<<endl;
8.      cout<<"f2: "<<f2<<endl;
9.      f4 = f1 = &f2;
10.     *f1 = 23;
11.     f4 = &f3;
12.     f3 = 17;
```

```
13.     cout<<"f1: "<<f1<<endl ;
14.     cout<<"f2: "<<f2<<endl;
15.     cout<<"f3: "<<f3<<endl;
16.     cout<<"++*f4: "<<++*f4<<endl;
  return 0;
}
```

Schritt	*f1	f2	f3	*f4	f5	Ausgabe
1			1		-	
2		23			-	
3					0x..von f2	
4	0x..von f2					
5		5				
6	0x..NULL					
7						f1: 0 (=NULL!)
8						f2: 5
9	0x..von f2			0x..von f2		
10		23				
11				0x..von f3		
12			17			
13						f1: 0x..von f2
14						f2: 23
15						f3: 17
16						++f4: 18 (17+1)

* = Dereferenzierungs Operator

“Multi” Pointer

Es ist möglich Mehrfach Pointer zu erstellen. Dies ist vor allem beim dynamischen Erstellen von Array über „Call by adress“ hilfreich.

```
int **p                 // Int **p enthält die Adresse von int *p, int *p enthält die Adresse von int p
```

Beim dreischichtigen Arrray muss dann also int ***p verwendet werden, beim vierschichtigen int ****p, usw.

Array

...

int a [x];

…

a [100] = 5;

…

Array mit Datentypen erzeugen

Dem Compiler muss vor dem Linken bekannt sein, wie viel Speicherplatz das Array braucht damit er diesen Speicherplatz reservieren kann weil die Speicherplätz für ein Array immer von 0 bis n direkt nacheinander reserviert werden. Aus diesem Grund muss die Variable für ein nicht dynamisches Array auch immer als Konstante definiert werden. Wenn die Grösse während der Laufzeit vergrössert wird sind die zusätzlichen Speicherplätz direkt nach den bestehenden aber diese können ja schon durch ein anderes Programm besetzt sein!

ACHTUNG:

Aufpassen, immer bei Index 0 starten beim Deklarieren sonst kann es passieren, dass es einen Speicherüberlauf gibt. C++ erkannt dies nicht und liest oder schreibt dann den Wert in einem Speicherbereich der vielleicht von einem anderen Code-Bereich belegt und somit auch beschrieben werden.

WICHTIG: Array Index sind immer unsigned, also im Wertebereich 0 - n

int a [110] ; Es werden 110 Integer Variablen erzeugt mit dem Index (Adresse) a [0] bis a [109]

Beispiel Speicheraufbau:

Variable Adresse	*Variable[Index]*	*Wert*	*Zugriff Variable*	*Speicher-reserviert?*
0Fx00000000	*a[0]*	*Wert Nummer 1*	*a[0]*	*Ja*
0Fx00000001	*a[1]*	*Wert Nummer 2*	*a[1]*	*Ja*
0Fx00000002	*a[2]*	*Wert Nummer 3*	*a[2]*	*Ja*
0Fx00000003	*a[3]*	*Wert Nummer 4*	*a[3]*	*Ja*
0Fx……..	*…*	*Wert Nummer n*	*…*	*Ja*
0Fx……..	*…*	*Wert Nummer n*	*…*	*Ja*
0Fx……..	*…*	*Wert Nummer n*	*…*	*Ja*
0Fx0000006D	*a[109]*	*Wert Nummer 100*	*a[109]*	*Ja*
0Fx0000006E	*a[110]*	*Wert Nummer 111*	*a[110]*	*NEIN*
0Fx0000006F	*a[111]*	*Wert Nummer 112*	*a[111]*	*NEIN*

Der rote Teil im Speicher ist beim Compilieren NICHT für das Array reserviert und somit kann es sein dass dieser Teil von Code für eine andere Anwendung reserviert ist. Der Speicherbereich ist jedoch für die Sprache C++ existent und somit kann auch darauf zugegriffen werden ohne dass ein Compiler-Fehler auftritt!

ACHTUNG: Die Grösser des Array muss beim starren Array als Konstante definiert werden. C++ bringt keinen Fehler wenn b nicht const gesetzt wird ABER dann könnte die Variabel verändert werden.

Beispiel Wert zuweisen:

```
main {
const int x (110);
int a [ x ];

a [ 100 ] = 5;                  // Dem Array-Index 110 den Wert 5 zuweisen
      }
```

Beispiel Wert zuweisen aber mit Programmierfehler:

```
main {
const int x (100);
int a [ x ];

a [ 100 ] = 5                   // Dem Array-Index 110 den Wert 5 zuweisen ABER da es nur den
                                // Index 0 bis 99 gibt ist dies ein Programmierfehler der keinen
                                // Compilerfehler ausgibt!
      }
```

Beispiele erzeugen Array mit Initialisierung:

```
int [ 12 ] = 51;                // 12 int mit Wert 51 erzeugen
```

Beispiele erzeugen Array mit mehrfacher Initialisierung:

```
int [9] = {1,2,3,4,5,6,7,8,9};
..oder..
int [ ] = {1,2,3,4,5,6,7,8,9}; // weil durch die Initialisierung bereits klar ist wie gross das
                                // Array ist.
```

Beispiele erzeugen Array mit Initialisierung:

```
char [ 12 ] = "Hallo";          // 12 char mit String „Hallo“ erzeugen
```

Call by value: Heisst der Wert der Variable wird übergeben!
Nachteil ist, dass auf diese Art kann NUR ein Wert aus der Funktion zurückgegeben werden!

Call by reference: Heisst die Adresse der Variable wird in einer Referenz Variabel gespeichert!
Vorteil ist, dass der Wert ohne dereferenzieren aufgerufen werden kann, jedoch ist die Referenz konstant der gleichen Variabel zugewiesen und kann nicht mehr verändert werden.

Call by adress: Heisst die Adresse der Variabel wird übergeben!
Vorteil ist, dass auf diese Art alle so viel gewünschte Variablen aus der Sequenz der Funktion gelesen oder geschrieben werden, wie der Code es verlangt.

Info zu Aufbau Array:

Variable a:	99	2	4	6	2	50
Index:	0	1	2	3	4	5
Adresse:	a+0	a+1	a+3	a+4	a+5	a+6

cout<<a[0]; Wert 99 von Array Variable a , Index 0 wird ausgegeben
cout<<a+0; Adresse F0xnnnn von Array Variable a , Index 0 wird ausgegeben

Wenn die Startadresse an die Funktion übergeben wurde, sind der Funktion automatisch ja auch alle weiteren Adressen bekannt, da diese beim Array erstellen im Speicher einfach im nächsten Speicherplatz gespeichert werden.

Beispiel für „call by refernce“:

```
#include <iostream>
using namespace std;
void function_array (int*);                 // Funktion deklarieren

void function_array (int *x) {              // Eintritt in die Funktion,  x ist ein  Pointer  welcher die Adresse
                                            // des ersten Index des Array  vom Aufruf (a+0) bekommt
x[0] = 12;                                  // Schreiben Wert 12 in den  a[0] über die Adresse (Pointer)
cout<<x[5];                                 // Ausgeben des letzten Wertes vom Array
x[3] = x[1];                                // Schreiben des Wertes aus Array a[1] in den a[3]
        }
_____________________________________________________________________________
int main () {
int a[6] = 0;
int b = a+0;
…
function_array (a+0);                       // Aufrufen der Funktion, Übergabe der Adresse vom Array a[0]
                                            // an den Pointer *x in der Funktion selbst
…
    }
```

Mehrdimensionales Array:

int a[8][8]; // 2 Dimensionale Array erstellen mit 8 x 8 Felder (Schachbrett)

0	0 = n	1 = n	2 = n	3 = n	4 = n	5 = n	6 = n	7 = n
1	0 = n	1 = n	2 = n	3 = n	4 = n	5 = n	6 = n	7 = n
2	0 = n	1 = 45	2 = n	3 = n	4 = n	5 = n	6 = n	7 = n

3	0 = n	1 = n	2 = n	3 = n	4 = n	5 = n	6 = n	7 = n
4	0 = n	1 = n	2 = n	3 = n	4 = n	5 = n	6 = n	7 = n
5	0 = n	1 = n	2 = 11 0x22fe2c	3 = n	4 = n	5 = n	6 = n	7 = n
6	0 = n	1 = n	2 = n	3 = n	4 = n	5 = n	6 = n	7 = n
7	0 = n	1 = n	2 = n	3 = n	4 = n	5 = n	6 = n	7 = n

cout<<a [2] [1]; // Ausgeben Wert **45** mit das Array Feld initialisiert wurde

int *p_a = NULL;
p_a = &a [5] [2];
cout<<p_a<<endl; // Ausgeben Adresse **0x22fe2c** vom Feld **5, 2**
cout<<*p_a<<endl; // Ausgeben Wert **11** mit das Array Feld initialisiert wurde

ACHTUNG:
* Dereferenzierungs Operator Pointer
[ ] Dereferenzierungs Operator Pointer
Dies bedeutet, dass ein Array auf Pointer aufgebaut ist!

float x[2][2]={{1,2},{3,4}}; // 2 Dimensionales Array erstellen und direkt initialisieren mit Werten

0	0 = 1	1 = 2
1	0 = 3	1 = 4

cout<<a[0][0]<<endl; // Wert **1** wird ausgegeben
cout<<a[1][0]<<endl; // Wert **3** wird ausgegeben

int a[8][8][8]; // 3 Dimensionale Array erstellen mit 8 x 8 x 8 Felder (Würfel)

int a[8][8][8][8]; // 4 Dimensionale Array erstellen mit 8 x 8 x 8 x 8 Felder (Würfel, wobei jedes Feld z.Bsp. 8 Attribute besitzt)

usw….

Grösse eine statischen Arrays ermitteln:
Achtung: Funktioniert nicht bei dynamischen Arrays!

Die Anzahl Felder eines Arrays kann anhand des Dividend von „Array Speicherplatz Gesamt“ (sizeof(Array)) und des „Speicherplatz ein Feld“ (sizeof(Array[0]) berechnet werden.

```
int main (){
const int x (100);
int testArray [x];
int sizeArray (0);

sizeArray =  sizeof(testArray)/sizeof (testArray [0]);

cout<<"\n\tGroesse Feld [0] Array: "<<sizeof testArray [0];    // Ausgabe = 4Byte
cout<<"\n\tGroesse Gesamtes Array: "<<sizeof(testArray);   // Ausgabe = 400Byte
cout<<"\n\tAnzahl Felder[n] Array: "<<sizeArray;                // Ausgabe = 100 Felder

        return(0);
}
```

Dynamisches Array

Um ein Array während der Laufzeit des Programmes zu erstellen, muss dieses dynamisch erstellt werden, sprich das Array wird erst während der Programmlaufzeit vom Code selbst erzeugt und der Speicher dazu erst zu diesem Zeitpunkt reserviert (Speicherplatz muss von Adresse 0 – n aneinandergereiht sein!). Dies macht das Programm zwar langsamer, jedoch kann dadurch das Array auf die richtige grösser erstellt werden was seinerseits wiederum Zeit einsparen kann.

Um ein mehrschichtiges Array mittels „Call by adress“ zu erstellen können sogenannte “Multi” Pointer (siehe unter Pointer) verwendet werden. Bei dieser Art von Pointer wird jeweils im „oberen“ Pointer die Adresse vom unteren Pointer enthalten, dass heisst man kann von oben direkt auf jede Schicht zugreifen.

Die Funktion um während der Laufzeit Speicher zu reservieren (= dynamisch) heisst **new** !
Dynamisch reservierter Speicher muss nach erlöschen der Gültigkeit selber wieder gelöscht werden! Der Befehl heisst **delete** !

Beispiel dynamisches 1 dimensionales Array:

```
void funktion (int);              // Deklarieren Funktion
void funktion (int *x){           // Adresse von Wert an Pointer x übergeben
x[2]=99;                          // 99 in Array wert[2] schreiben
….}

main (
int *wert=NULL;                   // Pointer erstellen und initialisieren
wert=new int[4];                  // Deklarieren dynamisches Array wert[4]

wert[0]=0;                        // Initialisieren Array wert[0]
wert[1]=0;                        // Initialisieren Array wert[1]
wert[2]=0;                        // Initialisieren Array wert[2]
wert[3]=0;                        // Initialisieren Array wert[3]

for (int i1=0;i1<3;i1++){
      wert[i1]=i1;}               // Initialisieren aller Array wert[0] bis [3] mit Wert von i1

funktion (wert);                  // Übergabe an Funktion

delete [] wert;                   // freigeben des dynamisch deklarierten Speicher
return 0;
)
```

Beispiel dynamisches 2 dimensionales Array:

```
void access(float **a){           // Erstellen von Pointer zweischichtig: **a enthält Adresse von *a, *a enthält
                                  // Adresse von Wert und mittels void wird Adressen vom beiden aus
                                  // Main übergeben
      cout<<a[0][0]<<endl;        // Ausgeben Wert 23 aus Zeile 0, Wert 0
      cout<<a[1][2]<<endl;        // Ausgeben Wert 5 aus Zeile 1, Wert 2
}

int main(){
```

```
        float **x;                  // Erstellen von Pointer zweischichtig: **x enthält Adresse von *x, *x enthält
                                    // Adresse von Wert

        x = new float*[2];          // In Pointer **x (referenziert wegen **) die Adressen von den zwei Zeilen
                                    // zuordnen. Gleichzeitig mit Funktion „new“ dynamisch Speicher während
                                    // Laufzeit vom Programm zuordnen.

        for(int i = 0; i < 2; i++){
            x[i]= new float[3];     // In Zeile i mit Funktion „new“ während Laufzeit des Programmes drei
                                    // Werte erzeugen
            }
        x[0][0] = 23;               // In Zeile 0, Wert 0 mit 23 initilaisieren
        x[1][2] = 5;                // In Zeile 1, Wert 2 mit 5 initilaisieren

        access(x);                  // Im Pointer die Adresse von Zeile 0 übergeben, in welcher die Adresse von
                                    // Wert 0 steht

        for(int i1=0;i1<2;i1++){    // Adresse der Zeilen anspringen
            delete x [3]; }         // Speicher der drei Spalten freigeben

        delete [] x;                // Speicher der Zeilen freigeben

        return 0;
}
```

		Wert Spalte 0	*Wert Spalte 1*	*Wert Spalte 2*
Zeile	**0**	**23**		
Zeile	**1**			**5**

Vektoren

Vektoren sind eine Klasse aus der Bibliothek <vector> was heisst sie muss mit dem Befehl #include <vectore> in das Projekt aufgenommen werden.

Die Syntax / Anweisung sieht wie folgt aus: vector <DATENTYP> NAME(ANZAHL);

```
Bsp: vector <int> stapel(10);      // Es werde 10 integer Elemente mit dem
                                   // Namen „stabel“ erstellt.
```

Zugriff funktioniert gleich wie beim Array:

```
stabel [6] = 99;          // Wert 99 wird in den Vektor Nr. 6 geschrieben
cout<<stabel [7];         // Wert im Vektor Nr. 7 wird ausgegeben
```

Mit dem Zusatz „at“ und den Runden Klammern kann die Gültigkeit des Vektor geprüft werden, was Sicherheit erhöht aber die Performance verschlechtert.

```
cout<<stabel.at(0);       //
```

Ebenfalls kann die Grösse des gesamten Vektors wie folgt abgefragt werden:

```
int size = vstabel.size(); // in der Variable size wird die Grösse gespeichert
```

Dyn. Speicher

Speicher während der Laufzeit des Programmes reservieren heisst, dynamischen Speicher erzeugen! Dies geschieht mit dem Syntax:

Dyn. einzelne Variablen:

```
int *testVariable;          // testVariable ist ein Pointer!
testVariable = new int;     // Speicher reservieren für testVariable
* testVariable = 9;         // Wert 9 dereferenziert in testVariable schreiben
..
..
delete testVariable;        // Freigeben von Speicher
testVariable = new int      // neuen Speicher reservieren für testVariable. Zuerst muss
                            // delete gemacht werden, weil der Pointer sonst nach der
                            // neuen Zuweisung die Adresse vom der ersten Zuweisung nicht
                            // mehr kennt und somit kann der Speicher nicht mehr freigegeben
                            // werden.
```

Auf jede Reservierung „new“ muss eine Freigabe „delete“ folgen sonst werden Speicherlöcher gebildet, welche während der ganzen Laufzeit vom Programm reserviert bleiben und erst nach dem beenden des Programmes wieder freigegeben wird. Im Schlimmsten Fall wie bei einem Betriebssystem heisst das, dass der Computer neu gestartet werden muss weil irgendwann kein Speicher mehr frei ist und das Programm abstürzt.

Dyn. Arrays

```
int *testArray;             // testArrayist ein Pointer!
testArray = new int [10];   // Speicher reservieren für testArray
* testArray [9] = 3;        // Wert 3 dereferenziert in testArray, Feld 9 schreiben
..
..
delete [ ] testArray;       // Freigeben von Speicher
testArray = new int [12];   // neuen Speicher reservieren für testVariable. Zuerst muss
                            // delete gemacht werden, weil der Pointer sonst nach der
                            // neuen Zuweissung die Adresse vom ersten zuweisen
                            // kennt und somit kann der Speicher nicht mehr freigegeben
                            // werden.
```

Datenstrukturen

Eine Datenstruktur ist eine Anzahl verschiedener Datentypen. Die Syntax sieht wie folgt aus:

```
struct testStruktur {       // Deklarieren Struktur
    string NAME;            // Deklarieren Variable 1
    int testInt;            // Deklarieren Variable 2
    double testDouble;      // Deklarieren Variable 3
    float testFloat;        // Deklarieren Variable 4
    …
};                          // Wichtig: mit ; abschliessen
```

Zugegriffen wird wie folgt mit dem Punktoperator, erstellt wird eine neue Struktur indem der Namen nach dem Variablennamen eingegeben wird.

```
testStruktur Struktur1;     // erstellen mit Namen „Struktur1“
testStruktur Struktur90;    // erstellen mit Namen „Struktur90“

Struktur90.NAME = “Hallo“;  // Zugriff auf String Variable NAME und initialisieren mit HALLO
Struktur1.NAME = “Welt“;    // Zugriff auf String Variable NAME und initialisieren mit Welt
Struktur90.testInt = 10;    // Zugriff auf integer Variable testInt und initialisieren mit Wert 10
…
```

Erstellen und initialisieren der Struktur gleichzeitig geht wie folgt:

```
testStruktur Struktur33 = {“Name“, “Vorname“, 99} ; // erstellen mit Namen „Struktur33“ und init.
```

WICHTIG: Es wird immer in der gleichen Reihenfolge nach dem die Variablen definiert werden auch die Variablen initialisiert.

Pointer auf Strukturen
Die Datenstruktur ist wie folgt definiert:

```
struct testStruktur {                   // Deklarieren Struktur
      string NAME;                      // Deklarieren Variable 1
      int testInt;                      // Deklarieren Variable 2
      double testDouble;                // Deklarieren Variable 3
      float testFloat;                  // Deklarieren Variable 4
      …
};                                      // Wichtig: mit ; abschliessen


testStruktur *pointerStruktur;          // deklarieren Pointer
testStruktur Struktur25;                // erstellen mit Namen „Struktur25“

pointerStruktur = &Struktur25;                  // Adresse Struktur 25 an pointer übergeben
(*pointerStruktur).NAME  = “Hallo“;             // Init. mit Punktoperator (alte Form)

pointerStruktur->NAME  = “Hallo“;               // Init. mit Pfeiloperator (neue Form)
```