Kurzanleitung zu C/C++

1. Datentypen

int, float, bool, char, size_t, double, long double. Diese können signed, unsigned und const sein. unsigned int ist das selbe wie size_t. (size_t muss jedoch mit stddef.h oder cstddef inkludiert werden. Bei letzterem ist std::size_t von Nöten.)

2. Präprozessor

Der C/C++ Compiler inkludiert erst die deklarierten Header Files und compiliert danach die einzelnen cpp Programme und übersetzt die als Objektdateien .o, um sie im Linker mit den benötigten Bibliotheken zu verbinden und aus ihnen das ausführbare Programm zu erstellen.

header.h -> #include/Praprozessor-Anweisung -> main.c/cpp -> Compiler -> main.o -> Linker (verknüpft mit Bibliothek) -> main.exe (oder main auf Linux)

3. Präprozessor-Anweisungen

```
#ifndef HEADER_H
#define HEADER_H
void foo(int c);
#endif
```

Diese im Header deklarierte Funktion kann später in mehreren Quelldateien mit #include "header.h" inkludiert werden (C's bzw C++'s eigene Header werden mit < > eingerahmt statt mit zwei Anführungszeichen). Die Funktion muss nur in einer Quelldatei definiert werden, da der Linker am Ende eh den Code in einer Datei zusammenfasst.

3.1. Globale Konstanten und Variablen

In Header Dateien werden Variablen ganz normal definiert, in der Quelldatei muss die Variable aber mit extern [datentyp] [variablenname]; deklariert werden.

Konstanten werden mit constexpr const [datentyp] [Konstantenname] = [Wert]; in der Header Datei definiert und mit extern const [datentyp] [Konstantenname]; in der Quelldatei deklariert und importiert. constexpr gibt es aber nur in C++, extern funktioniert aber in C wie auch C++. Der Sinn von constexpr ist eine Konstante zu haben, die wirklich nicht geändert werden kann, im Gegensatz zu einer const. Dies sorgt dafür, dass der Compiler schon beim Compilieren weiß, dass sich diese Konstante auch nicht ändern wird. Es funktioniert auch als Sicherung.

3.2. Lokale Variablen und Funktionen

```
static int x = 2; //die Variable ist lokal und kann nur in ihrer Funktion bzw Quelldatei verwendet werden
static int foo(); //die Funktion kann nicht von anderen Quelldateien außer der eigenen verwendet werden
```

3.3. Makro Definition

Makros sind die wahren Konstanten, die nicht geändert werden können, denn const Variablen sind nur begrenzt schreibgeschützt und können je nach Compiler mit Pointern geändert werden.

```
#define pi 3.14
float x = pi;
```

4. Namespaces

Man kann verschiedene Funktionen in C++ mit dem selben Namen haben, mann sollte diese aber dann in einen Namespace eintragen, um Konflikte mit anderen Funktionen zu vermeiden. Man deklariert einfach die Funktion innerhalb eines Namespaces und greift auf sie später mit dem selbst gewählten Namespace zu.

```
namespace foo {
void foof();
}
foo::foof(); //Zugrif auf foof aus foo namespace
using namespace std; //importiert std
```

5. Kommentare

// -> kommentiert den Rest einer Zeile
/* */ -> alles, was zwischen /* und */ steht, wird herauskommentiert.

6. Casting

Casting		
Erklärung	C	C++
Konvertierung zu int	(int) preis;	static_cast<int>(preis);

7. Binäroperationen

AND: 4 & 2 -> 0100 & 0010 -> 0000 -> 0
OR: 4 | 2 -> 0100 | 0010 -> 0110 -> 6
XOR: 6 ^ 2 -> 0110 ^ 0010 -> 0100 -> 4
NOT: ~6 -> ~0110 -> 1001 -> 9
Linksverschiebung(Verdopplung): << 2 -> <<0010 -> 0100 -> 4
Rechtsverschiebung(Halbierung): >> 2 -> >>0010 -> 0001 -> 1

8. Modulo

4 % 2 = 0 -> die Zahl ist rund
3 % 2 = 1 -> die Zahl ist ungerade

9. Pointer

```
int *p = &a; //Pointer p zur Adresse von int a
*p; //gibt den Wert aus, auf den p hinzeigt
const char* string = "ein string"; //ein Char-Zeiger, der auf einen konstanten C-String zeigt
Dieser string Pointer ist ein Char-Pointer, der auf einen Char-array deutet.
int * const pointer; //Dies wäre ein konstanter Pointer auf einen Integer. Ergo der Pointer ist unveränderbar und nicht sein Wert wie beim string oben (man liest die Deklarierung von rechts nach links)
```

In C++ gibt es des Weiteren noch den nullptr, was man Pointer zuweisen kann, die auf nichts zeigen sollen. (In C würde man NULL oder 0 verwenden)

10. Kommandozeilargumente

```
#include <stdio.h>
int main(int argc, char* argv[]) { //argc=Argumentezähler, argv=Array mit Argumenten
printf("Die Eingabe war: %s\n",argv[0]);
return 0;
}
```

Mit "./[programmname] hi" lässt sich das dann aufrufen.

11. Typedef

```
typedef [datentyp] [neuerdatentypsname];
[datentyp] foo = 3;
typedef [datentyp] [name][4]; //geht auch für arrays (oder auch pointer)
[name] arr = {1,2,3,4}; //Initialisierung
typedef struct [datentyp] {} [alias]; //und structs
```

11.1. Alias mit using

In C++ gibt es die Möglichkeit nicht nur mit typedef einen Alias für einen Datentyp zu definieren, sondern auch mit using.

```
using [alias] = [datentyp];
[alias] foo = 2;
```

12. Enums/Aufzählungen

Enummerations/Aufzählungen bzw enums sind selbst geschriebene Datentypen, dessen Werte programmintern nur eine Aufzählung der Werte in ihrer Reihenfolge darstellt.

```
//in C (einfache Version)
typedef enum { sternjaeger, transportschiff } Raumschiff; //komfortable Deklarierung
Raumschiff raumschifftyp = transportschiff; //transportschiff gibt 2 aus
//in C (ohne typedef)
enum Raumschiff { sternjaeger, transportschiff };
enum Raumschiff raumschifftyp = transportschiff; //hier wird enum bei der Deklaration gebraucht
//in C++
enum class Raumschiff { sternjaeger, transportschiff };
Raumschiff raumschifftyp = Raumschiff::transportschiff;
```

13. Random

```
#in C
#include <time.h>
#include <stdlib.h>
srand(time(NULL)); //generiert einen "zufälligen" Seed für rand
int r = rand % 30 +1; //generiert Zahl zwischen 0+1 und 29+1
#in C++
#include <random>
std::random_device rd; //generiert eine zufällige Zahl zwischen rd.min() und rd.max()
std::uniform_int_distribution<int> verteilung(0,1000); //erzwingt die Verteilung auf Werte zwischen 0 und 1000
int zufallszahl = verteilung(rd); //generiert Zahl von 0 bis 1000
```

14. Goto

```
for(int i = 0; i < 20; i++) {
for(int s = 0; s < 50; s++) {
if(i == 4 && s == 48) goto Labelname; //ist sehr praktisch, um schnell und einfach doppelte Schleifen zu beenden
}
}
Labelname:                                          //springt direkt in diese Zeile und beendet die Schleifen
```

15. Input und Output

```
#in C
#include <stdio.h>
int main() {
int a; scanf(%d,&a); //Wert der Eingabe wird an die Adresse von a geschickt
printf("%d0,a); //a wird in der Konsole ausgegeben
return 0;
}
#in C++
#include <iostream>
#include <string>
int main() {
int a; std::cin >> a; //a wird dem Wert der Eingabe zugewiesen
std::string c,s; std::getline(std::cin,c); std::cin.get(s); //speichert ganze Eingabezeile ab statt Leerzeichen zu überspringen
std::cout << std::to_string(a) <<" " << c << " " << s << std::endl; //gibt alles wieder in einer Zeile aus. Integer müssen zu strings konvertiert werden std::cerr << "Fehlermeldung" << std::endl; //Fehlermeldung im Fehlerkanal und nicht im normalen Ausgabekanal!
return 0;
}
```

15.1. Bemerkungen

Bei printf bedeutet %s->string, %d -> Variable, %p -> Pointer, %c -> Character.

16. Compilation

Für C:

```
gcc [C-Datei].c -o [programmname] && ./[programmname] Arraygröße  sizeof(feld)/sizeof(*feld)feld.size()
```

Für C++:

```
g++ [C-Datei].c -o [programmname] && ./[programmname]
```

17. Chrono

Dies gilt nur für C++. In C gibt es keine C eigene Methode Zeitintervalle zu messen.

```
#include <chrono>
std::chrono::time_point<std::chrono::steady_clock> start,ende;
start = std::chrono::steady_clock::now(); //Starter vom Timer
std::cout << "foo0; //ein beliebiger Befehl
ende = std::chrono::steady_clock::now(); //Ende vom Timer/der Stoppuhr
std::chrono::duration<double> vergangen = end - start;
std::cout << vergangen.count() << std::endl; //gibt vergangene Sekunden aus
auto millis = std::chrono::duration:cast<std::chrono::milliseconds>(vergangen); //konvertiert in Millisekunden
std::cout << millis.count() << std::endl; //gibt in Millisekunden aus
```

18. Malloc

Dynamische Speicheränderung		
Erklärung	C	C++
#include	<stdlib.h>	N/A
Erzeugen	char* s= (char*) malloc(sizeof(char)*4);	[datentyp]*p; p=new [datentyp];
sicheres Erzeugen	(char*) calloc(4,sizeof(char));	ist schon sicher
Speicherkopie	memcpy(sneu, s, sizeof(sneu));	[datentyp] neu_p = p;
Speicherkapazitätsänderung	(char*) realloc(s,2*sizeof(char));	arrayzeiger = new {datentyp}[anzahl];
Speicher löschen	free(s);	delete p;

19. Systembefehle

Dies geht in C wie auch C++. In C++ kann man jedoch auch std::system benutzen.

```
system("htop"); //ruft über die Kommandozeile des Betriebssystems htop auf
```

19.1. Bemerkungen

new -> ist eher für Objekte gedacht. Der pointer funktioniert nach der Datentypszuweisung wie der neue Datentyp und nicht wie ein Pointer. (Es ist auch möglich Arrays damit zu erzeugen)
calloc -> erzeugt im Gegensatz zu malloc einen Speicherpointer, der nur Nuller enthalten darf
memcpy -> fügt direkt in den neuen Speicherpointer ein, keine Zuweisung ist erforderlich (in C++ gibts auch std::memcpy)
realloc -> vereint in sich malloc, memcpy und free in einem

20. Grundlegende Programmstrukturen

20.1. if

```
if (!Bedingung && Bedingung || Bedingung) {
 ...
}
```

20.1.1. Auswahloperator

```
int a = 5 > 1 ? 23 : 1; //das selbe wie int a; if(5 >1) a= 23; else a = 1;
```

20.2. switch

```
switch(a) {
case 1:
 ...
break;
case 2:
 ...
break;
default:
 ...  }
```

20.3. While

```
while (Bedingung) {
 ...
}
```

20.3.1. do-while

```
do {
 ...
} while(Bedingung);
```

20.4. for-Schleife

```
for(int i = 0: i < 20; ++i) {
 ...
}
```

20.4.1. Direkter Zugriff auf alle Arrayelemente (geht nur in C++)

```
for(auto i : array) {
 ... }
```

20.5. Funktionen

```
int foo(int a) { // [datentyp] funktionsname([datentyp] a) { (als Datentyp kann man auch eine selbst definierte Klasse oder struct nehmen)
 ...
return [int-Wert]; // [datentyp] wird zurückgegeben (void braucht das nicht)
}
foo(5); // Funktionsaufruf
```

20.5.1. Funktionspointer

```
int add(int n, int m) { return n+m; } //Funktionsdefinition
int (*pointerzuadd)(int,int); // [Rückgabetyp (void bei void Funktionen)] [*funktionspointer] [funktionsparametertypen]; bzw der Funktionspointer wird erzeugt
pointerzuadd = &add; //Pointer wird Funktion zugeordnet
int sum = (*pointerzuadd)(2,3); //Aufruf
```

20.5.2. Übergabe per Referenz in Funktionen (nur in C++)

```
int add(int& c) { ... }; //in C++ wird so die Variable komplett übergeben statt wie in C erst einmal die Variablenadresse einem Pointer zu übergeben
```

20.5.3. Überladung von Funktionen

```
int quadrat(int i) {
return i*i;
}
double quadrat(double i) {
return i*i;
}
```

Nun ist quadrat() überladen und je nach dem welchen Datentyp quadrat() übermittelt, wird eine andere Funktion aufgerufen.

```
quadrat(2);   //   -> ruft int Funktion auf
quadrat(2.0); //   -> ruft jedoch double Funktion auf
```

20.5.4. Lambdas

Aufbau einer Lambda Funktion: [Liste aus Variablen, die aus der Quelldatei importiert werden] (Parameter für die Funktion, die beim Aufruf festgelegt werden) -> [datentyp, der ausgeworfen wird] { ... };

```
int r = 2; auto wo = [r](int a) -> int {return a*r;}; //wo ist ein Lambda Datentyp
wo(3); // Lambda-Funktionsausruf, gibt 6 aus
```

21. Zusammengesetzte Datentypen

21.1. Arrays

In C werden Arrays so erzeugt: int feld[{Arraygröße}] = {2,3 }; Oder zweidimensional: int cord[][] = { {2,4}, {5,6}};
Man greift auf ihren Index mit bspw feld[1] oder cord[1][0] zu. *feld zeigt auf den ersten Wert des Arrays, da der Arrayname wie ein Pointer zum ersten Arraywert funktioniert. Der Array ist aber kein Pointer!!!
In C++ benutzt man hingegen array<int,2> feld {{ 2,3 }}; , um das selbe feld wie in C zu erzeugen. Für cord würde man array< array<int, 2>,2> cord {{ {{2,4}}, {{5,6}} }}; schreiben. Es erzeugt, wie man sehen kann, einen Array im Array.
Den Zugriff macht man mit feld[1] oder feld.at(1). Bei cord: cord[1].at(1) oder klassisch cord[1][0]. Der Vorteil von at() ist, dass man einen Out-of-Bounds Fehler bei einem Zugriff auf nicht vorhandene Arrayelemente bekommt, wobei man beim klassischen Zugriff keinen Fehler sondern irgendeine Zahl an dieser Memoryposition erhält.

Array		
Erklärung	C	C++
Muss importiert werden	N/A	#include <array>
Deklaration und Definition	int feld[{Arraygröße}] = {2,3 };	array<int,2> feld {{ 2,3}};
zweidimensional	int cord[][] = { {2,4}, {5,6}};	array<array<int,2>,2> cord {{ 2,4}}, {{5,6}} }};
gibt 3 aus	feld[1];	feld.at(1);
gibt 5 aus	cord[1][0];	cord[1].at(0);
Arraygröße	sizeof(feld)/sizeof(*feld);	feld.size();
Arrayiteratoren	N/A	feld.begin() oder feld.end()
Array vergleichen	memcmp(feld1,feld2, sizeof(feld1));	std::equal

21.2. Bemerkungen

std::equal -> std::equal(std::begin(feld1), std::end(feld1), std::begin(feld2)) (benötigt <algorithm> und <iterator>)
memcmp -> (gibt, wenn sie gleich sind, 1 aus)

22. Strings

Achtung: Bei der Definition von Strings sollte man aufpassen: string x = "s" ist ein richtig definierter String. string x = ’s’ wäre jedoch ein string, dem ein Char zugewiesen wird!
C-Strings werden mit ’\0’ beendet.

Strings		
Erklärung	C	C++
Muss importiert werden	N/A (braucht für String-Funktionen <string.h>)	#include <string>
Deklaration und Definition	char s[] = "hi"; oder char* s = "hi";	std::string s = "hi";
gibt i aus	s[1];	s[1];
Stringlänge	strlen(s);	s.length();
Stringvergleich	strcpmp(s,"hi"); (bei gleichen wird 0 ausgegeben)	if(s == "hi")
String ändern	strcpy(s,"nicht hi");	s = "nicht hi";
Strings konkatenieren	strcat(feld1,feld2);(speichert in feld1)	s+="nichthi"; oder s.append("nicht hi");
Iteratoren	N/A	s.begin; s.end;
String zur Variable	atoi(s);	std::stoi(s); bzw stol,stof, stod und stoul
Substring	N/A	s.substr(3,5);
Substring finden	strstr(w1,w2);	s.find("hi);

22.1. Bemerkungen

find -> gibt Iterator aus, wo erstmals der regul. Ausruck auftaucht
strstr -> gibt Char pointer zum Anfang vom gefundenen w2 in w1 zurück
substr -> bei nur einem Argument gehts bis zum Ende durch