Kurzanleitung zu C/C++

1. Datentypen

int, float, bool, char, size_t, double. Diese können signed, unsigned, short, long 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
```

Des Weiteren behält x seinen Wert bei, auch nachdem man ihren Block verlassen hat, weswegen ein ++x; später mehr als 3 sein kann.

Das Gegenteil von static ist auto, was der Compiler standardmäßig jeder Variable zuordnet. Wenn der Funktionsblock verlassen wird, wird auch der Wert von auto x = 2; gelöscht.

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)

9.1. Referenzen (nur bei C++)

```
int x = 24;
int& ref = x; //eine Referenz agiert wie eine normale Variable, statt einem Pointer gleich, kann aber nie NULL sein
std::cout << ++ref<<"\n"; //gibt 25 aus
```

9.2. Smart Pointer (nur bei C++)

Smart Pointer haben den großen Vorteil zu rohen Pointern, die manuell mit new erzeugt wurden, dass sie automatisch nach ihrer Laufzeit gelöscht werden und es so zu keinen Memoryleaks kommen kann. (normale rohe Pointer werden aber nach ihrer Laufzeit gelöscht, wie alle anderen Variablen auch. new braucht nämlich immer ein delete, sonst gibts einen Memoryleak (bei malloc ist es genauso))
Smart Pointer funktionieren nach dem RAII-Prinzip = "Resource Acquisition Is Initialization". Das heißt, dass der Konstruktor eines Objekts ihm Speicher zuweist und sein Destruktur diesen zugewiesenen Speicher später befreit.

9.2.1. Shared Pointer

```
#include <memory>
std::shared_ptr<int> ptr = make_shared<int>(42); //ptr zeigt auf 42
(*ptr)++;  //nun auf 43
auto ptr2 = ptr; //ptr2 zeigt auf den selben Wert wie ptr
ptr.use_count(); //ptr weiß, dass es einen weiteren shared Pointer gibt, der auf 43 zeigt, also gibt er zurück, dass zwei gleiche pointer existieren
auto ptr3 = ptr2; //bei use_count() wird nun 3 ausgegeben, da alle shared Pointer der gleichen Adresse voneinander wissen
ptr2.reset(); //ptr2 wird gelöscht, nun sinds nur noch 2
int* i = ptr.get();  //ein roher Pointer kriegt den Wert von ptr. use_count() ergibt aber trotzdem 2, weil i roh ist
```

9.2.2. Unique Pointer

```
std::unique_ptr<int> ptr = make_shared<int>(42); //ein Unique Pointer. Er funktioniert fast wie ein shared Pointer
auto ptr2 = ptr; //nur ist der gravierende Unterschied, dass er keine Smartpointer neben sich duldet! Diese Zeile wär ein Fehler
```

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.
Anmerkung: statt argv[] kann man als Funktionsparameter oben auch char **argv verwenden, da Arrays als Funktionsparameter als Pointer vom Compiler behandelt werden. (genauso wie Funktionen als Parameter)

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. Dateien schreiben und lesen

in C

```
#include <stdio.h>
FILE *pF = fopen("lesen.txt","r"); //"r" = read
char linebuffer[255]; //maximaler Linebuffer
//Durchlesen
while(fgets(linebuffer,255,pF)!= NULL) printf("%s",buffer);
//gibt Zeilen aus, solange es Zeilen zu lesen gibt. jedes Mal, wenn fgets aufgerufen wird, wird eine weitere Zeile gelesen
fclose(pf); //Filepointer wird geschlossen
//Ausgabe
FILE *pf = fopen("schreiben.txt","w"); //"w" = (Über)schreiben und "a" = anfügen
char *string = "nur ein c string\n";
fprintf(pf,string); //bei jedem Aufruf wird eine neue Zeile geschrieben bzw an eine Text angefügt
fclose(pf); //schließt den Filepointer wieder
```

in C++

```
#include <fstream>
std::ifstream quelle("speicher.txt");
if (!quelle.good()) {  //überprüft, ob es möglich ist die Datei zu öffnen
cerr << "Datei konnte nicht geöffnet werden\n"; return 1;
}
//Durchlesen
while(quelle.good()) { std::string s; quelle >> s; std::cout << s << "\n"; } //jede Abfrage geht eine Zeile weiter
//Ausgabe
std::ofstream aus("aus.txt");
if (!quelle.good()) {
cerr << "Datei konnte nicht geöffnet werden\n"; return 1;
}
std::array<std::string,3>t {{ "hi","ein string","noch ein String!" }};
while(quelle.good()) { for(auto i : t) aus <<i<<"\n";} //fügt den Array Zeile für Zeile ein
```

16.1. Filesystem (nur für C++)

```
#include <filesystem>
std::string path = "/home/notebook";
for(const auto& eintrag : std::filesystem::directory_iterator(path)) std::cout << eintrag.path() << std::endl;
//gibt jede Datei bzw jeden Ordner im Ordner dieses Pfades aus
```

17. 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]
```

18. 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
```

19. Malloc

Jede manuelle Speicherzuweisung braucht immer ein free() bzw delete, da es sonst zu Memoryleaks kommt, weil C++ diese Zuweisungen nicht rückgängig macht.

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;

20. 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
```

20.1. Bemerkungen

int * var = malloc(sizeof(int)*300); bzw int* var = new int[300]; -> der Pointer var ist auf dem Stack, weist aber auf einen zugewiesenen Speicherplatz auf dem Heap, welcher nicht wie der Stack später automatisch freigemacht wird.
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

21. Grundlegende Programmstrukturen

21.1. if

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

21.1.1. Auswahloperator

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

21.2. switch

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

21.3. While

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

21.3.1. do-while

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

21.4. for-Schleife

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

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

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

21.5. 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
```

21.5.1. 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
```

21.5.2. 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
```

21.5.3. 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
```

21.5.4. 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
```

21.6. Try und catch (nur C++)

Einfach gesagt ist throw dazu da einen Fehler zu übergeben und catch im try-catch-Block sorgt dafür, dass das Programm weiterlaufen kann, aber auf den Fehler eingeht.

```
float divide(float f1, float f2) {
if f2( == 0) throw "Division durch Null!"; //Dieser String wird von catch später aufgefangen
else return f1/f2;
}
try { divide(2.0/0); }
catch(std::string err) { std::cerr << err << std::endl; } //Fehler wird ausgegeben
```

22. Templates (nur C++)

Templates machen es möglich, dass Funktionen mit verschiedenen Datentypen verwendet werden können.

```
template<typename T>
bool kleiner(const T& a, const T& b) { //Funktion ist für jede Parameterart gültig
return a < b;
}
```

Wenn man vermeiden will, dass die Funktion für bestimmte Datentypen verfügbar ist, überlädt man die Templatefunktion.

```
template<bool>
bool kleiner(bool a, bool b) {
static_assert(false, "sinnloser bool-Vergleich"); //gibt Fehlermeldung
return 0;
}
```

23. Zusammengesetzte Datentypen

23.1. 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

23.2. 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)

24. 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 und funktionieren wie C-Arrays, da sie eigentlich Char-Arrays sind.
C++-Strings können mit insert und erase bearbeitet werden. (ähnlich wie ein Vector)

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);

24.1. 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

25. Structs

Structs sind C’s Art Objekte bzw eigene Datentypen zu erzeugen (structs haben jedoch keine Methoden). Der Unterschied zwischen C’s Structs und C++’s structs ist, dass die structs von C++ die Möglichkeit haben, mit Konstruktoren ininitialisiert zu werden. Des Weiteren muss man in C++ structs nicht mit struct initialisieren, was in C nur in Kombination mit typedef geht.
In C++ kann man structs übrigens auch einen Destruktor mit ˜[Structname]() {} geben.

```
struct character {     //für typedef: typedef struct { char* name; }character;
char* name;
character(char* a) : name(a) { [...] }; //Der Konstruktor geht nur in C++, in C ist das ein Fehler
}
struct character roboter;   //in C++ geht auch character roboter;
struct *character rpointer = &roboter;  //in C++ geht auch nur *character rpointer = &roboter;
rpointer->name = "R2-D2";  //das selbe wie (*rpointer).name = "R2-D2";
//in C++ kann man das alles auch mit dem Konstruktor verkürzen
character roboter("R2-D2");
//in C und C++ kann man auch Arrays mit struct erzeugen
struct character roboter[];
```

25.1. Unions

Unions sind eigentlich wie Structs, nur können sie lediglich eine einzige Variable gleichzeitig haben.

```
union character {     //für typedef: typedef union { char* name; }character;
char* name; int version;
character(char* a) : name(a) { [...] }; //Der Konstruktor geht nur in C++, in C ist das ein Fehler
}
union character roboter;   //in C++ geht auch character roboter;
union *character rpointer = &roboter;  //in C++ geht auch nur *character rpointer = &roboter;
rpointer->name = "R2-D2";  //das selbe wie (*rpointer).name = "R2-D2";
rpointer->version = 4; //nun ist der Name weg
//in C++ kann man das alles auch mit dem Konstruktor verkürzen
character roboter("R2-D2"); //oder character(roboter(4);
//in C und C++ kann man auch Arrays mit unions erzeugen
union character roboter[];
```

25.1.1. Structs und unions zusammen verwenden

Unions kann man dazu verwenden, dass nur bestimmte Variablen besetzt werden und andere nicht, um Platz im Speicher zu sparen.

```
struct character {
char *name;
union { char *system; int alter; }
} struct character han_solo;
han_solo.name = "Han Solo"; han_solo.alter = 28;
struct character r2d2;
r2d2.name = "R2.D2"; r2d2.system = "System 73";
```

26. Vector (nur C++)

Ist wie ein Array, nur kann der Vector in seiner Laufzeit auf dem Memorystack vergrößert oder verkleinert werden. Der Zugriff funktioniert beim Vector genauso wie beim Array. liste[3] = 2; oder liste.at(3) = 2;

```
#include <vector>
std::vector<int> liste {1,2,3};
liste.push_back(4); // liste = {1,2,3,4}
liste.insert(liste.begin(),0); // liste = {0,1,2,3,4}
liste.erase(liste.begin()+2); //liste = {0,1,3,4}
liste.pop_back(); //liste = {0,1,3}
```

27. Maps (nur in C++)

Maps ordnen einem Schlüssel bzw einem Eingabewert einen Ausgabewert zu. Des Weiteren sind sie auch sortiert.

```
#include <map>
map<char, int>karte { {'a',2}, {'c', 5} };
karte.insert(make_pair('b',3));
karte['r'] = 6;   //einfacherer Insert, der bei bestehenden Einträgen zum Update führt
std::cout << karte'a'] << std::endl; //Zugriff
karte.erase('r'); //löscht Eintrag für 'r'
karte.clear();  //löscht alles in der Karte
karte.empty()   //gibt true aus, wenn die Karte leer ist
```

27.1. unsortierte Map

Die unsortierte Map ist eine Hashmap. Alle ihre Schlüssel werden mit einem Hash berechnet. Außerdem ist sie unsortiert, wie der Name auch schon sagt. Sie kann im Gegensatz zu einer map schneller sein, da ihre Einträge unsortiert sind.

```
unordered_map<char, int> hash_map { {'a',2 }, {'c',5} };
```

Ihr Zugriff auf sie funktioniert genauso wie bei einer normalen Map.

28. Softwaretests mit assert und static_assert

Mit assert kann man bei der Laufzeit eines Programmes feststellen, ob eine Bedingung erfüllt ist. Sofern diese nicht erfüllt ist, beendet sich das Programm und es kommt zur Fehlermeldung. static_assert macht so ziemlich das selbe, jedoch nur zur Compilierzeit und verhindert die Compilierung, sofern die Bediningung nicht eintrifft. static_assert ist nur Teil von C++.

```
#include <assert.h>
assert(a == b); //ist a nicht gleich b, wird das Programm beendet
std::static_assert(a == b,"A ist ungleich B"); //ist a nicht gleich b, geht die Kompilierung schief
```

29. Klassen (nur in C++)

Klassen sind structs ziemlich ähnlich. Der einzige Unterschied ist, dass man mit ihnen Daten kapseln kann und ihnen Methoden geben kann. Wie auch, dass man sie von anderen Klassen erben lassen kann und somit ihnen die Methoden und Variablen anderer Klassen übergibt, die sie jedoch für sich überschreiben können.

```
class Mensch{
public:
        void hallo() { std::cout << "Hallo\n"; }
};
class Hans : public Mensch {   //erbt von der Klasse Mensch
public:         std::string nachname;          Hans(std::string n) : nachname(n) { //bzw this->nachname = n;
        }
        std::string getnachname() { return nachname; } //ohne this-> geht auch
        ~Hans() { std::cout << "Ade, schöne Welt\n";  } //Destruktor
private:
        std::string versicherungsnummer; //Privat! Darauf hat man keinen Zugriff
};
Hans derHans("Meier");
derHans.getnachname(); //gibt nachname zurück
derHans.hallo(); //gibt die geerbte Methode von Mensch zurück
```

29.1. Polymorphie

Klassen sind polymorph, wenn sie eine Klasse haben, von welcher sie alle erben. Man nehme als Beispiel es gebe die Klassen Hans, Klaus und Erwin. Alle drei erben von der Klasse Mensch, somit kann man einen Array aus ihnen erzeugen, der den Datentyp Mensch hat: Mensch menschen[] = { Hans, Klaus, Erwin };

29.2. Operatorenüberladung

In C++ ist es möglich die Operatoren von Klassen zu überladen, indem man sie als Methoden der Klasse definiert.

```
class Hans : public Mensch {   //erbt von der Klasse Mensch
public:         std::string nachname;         Hans(std::string n) : nachname(n) { //bzw this->nachname = n;
        }
        Hans operator+(const Hans &andererhans) { std::cout << nachname <<" grüßt den " <<
andererhans.nachname <<std::endl;
Hans unwichtig("hi"); return unwichtig; //Die Operatormethode erfordert einen Rückgabewert
} // + wird überladen
        std::string getnachname() { return nachname; } //ohne this-> geht auch
        ~Hans() { std::cout << "Ade, schöne Welt\n";  } //Destruktor
private:
        std::string versicherungsnummer; //Privat! Darauf hat man keinen Zugriff
};
Hans derHans("Meier");
Hans andererHans("Schröder");
derHans + andererHans; //Meier grüßt den Schröder und unwichtig wird ausgegeben, aber nicht verwendet
```

30. Programmierkonzepte

30.1. Memory/Speicher

Der Speicher ist das, was der Computer braucht, um Daten für eine gewisse Zeit im Arbeitsspeicher (RAM = Random Access Memory) zu speichern.

30.2. Cache

Dies sind die Daten, die die CPU später häufiger verwenden will, diese sind in der static RAM (SRAM) gespeichert bzw teilweise in der CPU.

30.3. Buffer/bzw zu Deutsch Puffer

Teil des Arbeitsspeichers, der benutzt wird, um von der CPU schnell Daten abzuspeichern und wieder einzulesen.

30.4. CPU

CPU (Central Processing Unit) bzw zu Deutsch der Prozessor macht alle Rechenoperationen des Rechners und kommuniziert dabei mit dem Halbleiterspeicher (RAM) über die schenelle Datenleitung BUS.

30.5. Stack

Speicherungskonzept der Informatik, bei dem Daten auf einen Stapel gelegt werden und nach LiFo (Last in First out) abgearbeitet werden.
Bzw Teil des Speichers: Er ist typischerweise ein bis zwei Megabyte groß und enthält die Daten, die das Programm speichert und ausliest. Wird der Stack zu groß, kommt es zum Stackoverflow (bspw wenn eine Liste unendlich lang größer wird/bzw ein Array bzw Liste zu groß ist oder das Programm zu viele Operationen auf den Stack legt)
Jede Variablen bzw Pointerdefinition oder Funktionsaufruf legt eine Operation auf den Stack, der von dem Prozessor abgearbeitet wird. Wenn man mehr Arbeitsspeicher benötigt, lädt man das benötigte Datenobjekt in den Heap mit malloc oder new, muss diesen aber später selber freimachen.

30.6. Heap

Der Heap bzw Haufen kann nach beliebger Reihenfolge erweitert werden und ist der größere Teil des Speichers. LiFo gilt nicht wegen des wahlfreien Zugriffs des Heaps.
Er ist im Gegensatz zum Stack dynamisch mit malloc oder new erweiterbar, wird aber weil er nicht stackgleich abgearbeitet wird, nicht von selber gelöscht und muss manuell freigemacht werden.

30.7. Array

Bzw zu Deutsch Feld, ist eine Datenstruktur, bei der jedes Element nach seiner Speicheradresse benachbart ist und dessen endgültige Größe schon bei der Initialisierung endgültig ist, damit diese Speicheradressen auch schon vorreserviert sein können.

30.8. Dynamischer Array

Es ist ein Array, der wenn er mehr Platz benötigt, sich einen neuen größeren Array erzeugt und seinen Inhalt dort hineinkopiert.

30.9. Verkettete Liste

Eine Datenstruktur, bei der jedes Element einen Zeiger auf das nächste Element enthält. -> dies ist die einseitige Kette bzw Linked list
Die doppelt verkettete Liste funktioniert nach dem Prinzip, dass jedes Element, welches weder das erste noch das letzte Element ist, einen Zeiger zum Vorgänger und Nachfolger enthält.