1 C++ AVR Skriptum

Table of Contents

Tolle Unterlagen für C++ mit interaktiven Beispielen :
https://en.cppreference.com/w/cpp/language/for

1.1 Begriffe

Überladung
Namespaces
Überladung
Referenzen
OOP: Klassen, Vererbung
generische Programmierung: Templates
Operator Overloading
Polymorphie
Inline Functions
Makros

Der AVR C++ Compiler

Viele Eigenschaften von C++ fehlen im AVR Gnu C++ Compiler (G++.exe)

In diesem Skriptum sollen die Funktionen, die in den Libraries von Arduino vorkommen, beschrieben werden.

1.1.1 Deklaration und Definition

Deklarationen erklären dem Compiler, welche Variablen/Funktionen benötigt werden. Deklarationen dürfen beliebig oft wiederholt werden.

Wenn eine Deklaration vorhanden ist kann eine Variable/Funktion verwendet werden, auch wenn sie nicht definiert ist, solange sie nicht aufgerufen wird.

Definition legt die Variable/Funktion an und belegt dadurch Speicher (die Variable „lebt“)

Kontrollfrage: Was ist Deklaration? Was ist Defininition? Wie Deklariert man eine Variable?

Deklaration einer Funktion

void test(); und extern void test(); sind gleichbedeutend

Definition einer Funktion

void test(){}

Deklaration einer Variablen

extern int x;

Definition einer Variablen (und gleichzeitige Deklaration)

int x;

1.1.2 Globale Variable / Extern

Globale Variable, die über mehrere Quelltext-Dateien übergreifen, vereinbart man in einem gemeinsamen Headerfile als extern. Externe Variable werden am Heap angelegt und leben daher bis main beendet wird.

Kontrollfrage: Wie vereinbart man eine globale Variable, die in mehreren Quelltextdateien verwendbar ist?

main.cpp

```
#include "header1.h"
int globalVar;
int main(){
 globaVar = 5;
 test();
 int x=globalVar;
 int aha = globalVar; // aha == 0
}
```

header1.h

```
#ifndef HEADER1_H_
#define HEADER1_H_

extern int globalVar;

void test(){
    globalVar = 0;
}
#endif /* HEADER1_H_ */
```

1.1.3 Static Variable

Werden am Heap angelegt und leben daher so lange wie das Programm läuft. Die Sichtbarkeit dieser Variablen ist aber lokal

Kontrollfrage: Wozu dienen static Variable?

```
#include <avr/io.h>

int testit(){
    static int x = 5;   // Initialisierung wird nur einmal ausgeführt!
    return ++x;         // x funktioniert wie eine globale Var, ist aber nur
                        // innerhalb der Funktion sichtbar
}

int result;

int main(){
      for (uint8_t i=0; i<2;i++){
            result = testit();
      }
}

//result == 7 !
```

1.1.4 Überladung

Funktionen dürfen mit gleichen Namen deklariert werden, solange sie sich in der Parameterliste unterscheiden (gilt nicht für den Rückgabetyp).

Kontollfrage: Was ist Überladung?

```
void test() {}
void test(int x) {}
int main()
{
     test();
     test(1);
}
```

1.1.5 Bedingte Compilierung

Kontrollfrage: Wozu dient Bedingte Compilierung?

Ein Bereich des Quelltextes wird nur dann verwendet, wenn eine Bedingung erfüllt ist (z.B: ein Symbol definiert #ifdef oder nicht definiert #ifndef ist.

Beispiel: beim ersten Aufruf ist das Symbol „Arduino_h“ noch nicht definiert, der Quelltext wird eingebunden; beim nächsten Mal ist dann das Symbol definiert und der Quelltext wird kein zweites Mal verwendet.

#define DEBUG

#undef TRACE

```
#ifndef Arduino_h
#define Arduino_h
  … hierhier kommt der Quelltext
#endif


#ifdef TRACE
   … Tracing Code here
#endif

#ifdef DEBUG
   … Debug Code here
#endif
```

1.1.6 CPP Files aufteilen

Mehrere CPP Files werden automatisch zusammengebunden, wenn sie im gleichen Ordner liegen: im folgenden Beispiel liegen main.cpp und cpp1.cpp im gleichen Ordner. Das Programm main.cpp compiliert fehlerfrei.

main.cpp

```
test();
int main(void){
 test()
}
```

cpp1.cpp

```
void test(){
      for (int i = 0 ; i< 100; i++);
}
```

1.1.7 C Header Files in C++ Code einbinden "extern C "

```
extern "C" {
  #include "my-C-code.h"
}
```

Beispiel:

```
extern "C" {
 #include "c1.h"
}

int main(void)
{
 test();
}
```

c1.h

```
void test(){


}
```

Besser so: Der normale C Compiler kennt das Symbol __cplusplus nicht und ignoriert daher das „extern{}“

main.cpp

```
#include "c1.h"
int main(void)
{
      test();
}
```

c1.h

```
#ifdef __cplusplus
extern "C"{
#endif

void test(){
            volatile int i; i++;
      }
#ifdef __cplusplus
} // extern "C"
#endif
```

1.1.8 Namespaces

Programme werden von mehreren Programmierern entwickelt die jeder seine eigenen Symbolnamen vereinbart. Um Kollisionen zu vermeiden, wird ein Namensraum vereinbart. Die Funktion heißt dann nicht func(), sondern first_space::func() und kollidiert damit nicht mit einer func() im zweiten Namensraum.

```
Kontollfrage: Wozu dienen  Namespaces und wie vereinbart man einen Namespace?
```

```
#include <iostream>
using namespace std;

namespace first_space {
   void func() {
      cout …
   }
   using Print::write;  // verwende in weiterer Folge die Funktion write() aus dem
                        // Namespace Print

   void abc(){
       write('A');      // anstelle Print::write('A')
   }
}

namespace second_space {
   void func() {
   }
}

int main () {
   first_space::func();  // :: scope operator
   second_space::func();
}
```

1.1.9 Default Parameter

```
void f(int x = 3, int y = 4);

f(5,2); // Aufruf f(5,2)
f(1);   // f(1,4)
f();    // f(3,4)
```

1.1.10 Referenzen

1. kürzerer oder verständlicherer Aliasname für ein bereits bestehendes Objekt/Variable
2. zur Optimierung, um Kopien von Objekten zu vermeiden

Kontrollfrage: Wie übergibt man in C++ Rückgabeparameter an Unterprogramme

Parameterübergabe an Unterprogramme; über diesen Parameter kann ein Wert zurückgegeben werden:

```
void func1(uint8_t & var){   // var hat die gleiche Adresse wie i am Stack
   var = 5;
}
void loop() {
   uint8_t i;
   func1(i);  // i==5
}
```

1.1.11 OOP: Klassen, Vererbung

Siehe Kapitel OOP in C++

- Klassen und Objekte
- Abstraktion
- Datenkapselung
- Information Hiding
- Vererbung
- Polymorphie

1.1.12 Generische Programmierung: Templates

Generische Programmierung ist ein Verfahren zur Entwicklung wiederverwendbarer Software-Bibliotheken. Dabei werden Funktionen möglichst allgemein entworfen, um für unterschiedliche Datentypen und Datenstrukturen verwendet werden zu können.

Programme werden unabhänig von einem bestimmten Datentyp.

Kontrollfragen: Was ist generische Programmierung, welchen Vorteil bietet sie?

Wie schreibt man in C++ ein Template?

```cpp
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& AddIt (T const& a, T const& b) {
   return a+b;
}

int main () {
   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

1.1.13 Operator Overloading

folgende Operatoren können überladen werden:

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

Beispie1

```
#include <avr/io.h>
typedef enum Tcolor {red, blue, green, dark} ;
Tcolor operator- (Tcolor x) {
      return dark;
}
int main(void)
{   // turn off color by using - operator
    Tcolor c1 = - blue;
}
```

Beispiel2 (Fortgeschritten)

```
class Ta{
      public:
            int sum;
            Ta(int sum=0){  //Default Parameter
                  Ta::sum = sum;
            }
            Ta& operator+ (Ta summand2){   // wie ein Funktionsname „operator+()“
                  sum=sum+summand2.sum;
                  return *this;     // folgt dem this-Zeiger
                                    // und gibt daher sich selbst
                                    // als Referenz zurück
            }
};

int main(void){
      Ta A(3);
      Ta B(4);
      Ta C(1);
      Ta Summe;          //ruft Konstruktor mit default-Parameter auf
      Summe=A+B+C;       // A addiert B und kriegt A zurück A=A+B,
                         // dann A = A+C und schließlich Summe=A
                         // ist schöner zu schreiben als
                         // Summe=(A.operator+(B)).operator+(C);
      int i = Summe.sum;
}
```

1.1.14 Inline Functions

Der Code einer INLINE-Funktion wird überallhin kopiert, wo die Funktion aufgerufen wird (es erfolgt also kein Unterprogrammaufruf) (vgl. Macros in C)

Kontrollfrage: Was ist eine Inline-Funktion, welchen Vorteil hat sie?

Wie vereinbart man eine Inline-Funktion?

```
inline int Max(int x, int y) {
   return (x > y)? x : y;
}
```

Member-Methoden sind implizit inline

```
class TClass{
  public:
      int memberfunction(){
            return 0;
      }
}
```

1.1.15 Macros

Macros sind Textersetzungen, die auch wie eine Funktion Parameter haben können

```
#define F_CPU 1000000UL
#define BAUD 38400
#define UBRR_VALUE (((F_CPU) + 4UL * (BAUD)) / (8UL * (BAUD)) -1UL)

#define DELAY_SPI(X) { int ii=0; do { asm volatile("nop"); } while (++ii <
(X*F_CPU/16000000)); }

#define bit(b) (1UL << (b))

void main(){
    uint8_t mask = bit(3);   //vgl (1<<3)
}
```

Achtung auf die Klammersetzung!
https://manderc.com/preprocessor/definedirective/index.php

1.1.16 Typedef

Wie eine Textersetzung mit #define, aber wird vom Compiler überprüft

```
typedef unsigned int word;

word nr = 12000;
```

1.1.17 Function-Pointer

Unter Verwendung von typedef kann eine Funktion über einen Pointer wie folgt aufgerufen werden:

```
int add(int a, int b){
      return a+b;
}
int sub(int a, int b){
      return a-b;
}

void main() {
  typedef int (*myFunctionPointer)(int,int);
  myFunctionPointer p[2] = {&add, &sub}; //Addresses of the functions
  int sum = p[0](1,5);
  int dif = p[1](3,4);
}
```

1.2 OOP in C++

- Klassen und Objekte
- Abstraktion
- Datenkapselung
- Information Hiding
- Vererbung
- Polymorphie

Kontrollfragen:

Was ist ein Objekt, was ist eine Klasse?

Was ist Abstraktion?

Was ist Datenkapselung?

Was ist Info Hiding?

Was ist Polymorphie?

1.2.1 Was ist ein Objekt? Was ist eine Klasse?

Klasse: die Beschreibung, (vgl. Typ) Beispiel: Mensch

Objekt, Instanz: die Realisierung, (vgl. Variable) Beispiel: Sepp

Kontrollfrage: Braucht eine Klasse Speicherplatz?

AW: nein, sie ist nur eine Beschreibung, Speicherplatz braucht ein Objekt

Abstraktion: die Objekte werden der Realität nachgebaut (z.B: TCar, TPoint usw)

Datenkapselung: alle Eigenschaften und Methoden (= Functions) sind in der Klasse eingeschlossen

Information Hiding: manches wird für die Öffentlichkeit versteckt (Schutzklasse private, protected, public); private Eigenschaften können über setter/getter zugegriffen werden

```
class TCar {
      private:
            int Speed;
      public:
            void drive();
            void setSpeed(int);
            int getSpeed();
};
```

1.2.2 Vererbung

Alles was in TBeing public ist bleibt public

```
#include <avr/io.h>
class TBeing {                      // Basisklasse
   public:
      int  legs;
      void run(){}
};
class THuman:public TBeing {        // Abgeleitete Klasse, Subklasse, Kindklasse
   public:
      void speak(){};
};

int main(){
      TBeing be;                    // Instantiierung, Instanzierung
      THuman otto;
      otto.speak();                 // Methodenaufruf
      otto.run();
      otto.legs = 2;
}
```

1.2.3 Konstruktor

Wenn kein Konstruktor angegeben wird, dann wird ein Defaultkonstruktor aufgerufen.

```
#include <avr/io.h>
class TBeing {
   protected:                                  // Zugriff für Kindklassen erlauben
      int legs;
   public:
    TBeing(int nr){ legs = nr; }
      void run(){}
};

class THuman:public TBeing {
   public:
    THuman(int nr):TBeing(nr){}                // Aufruf des Konstruktors Basisklasse
      int getLegs(){return legs;}              // Zugriff auf legs wegen protected
      void speak(){};
};

int nr;
int main(){
      TBeing be(4);           // 4 legs
      THuman otto(2);         // 2 legs
      otto.speak();
      otto.run();
      nr = otto.getLegs();
}class THuman:public TBeing {
   public:
    THuman(int nr):TBeing(nr){}                // Aufruf des Konstruktors Basisklasse
      int getLegs(){return legs;}              // Zugriff auf legs wegen protected
```

```
        void speak(){};
};
```

1.2.4 Initialization Lists to Initialize Fields

Wird benötigt für die Initialisierung von Konstanten, Referenzen und wenn es keinen Default-Konstruktor für eine Basisklasse gibt.

Syntax

```
class Class123{
    private:
                double X, Y, Z;
    public:
                Class123( double a, double b, double c): X(a), Y(b), Z(c) {

               }
};
```

hier könnte man auch schreiben

```
                Class123( double a, double b, double c) {
                             X=a;Y=b;Z=c;
               }
```

aber das funktioniert nicht bei Konstanten und Referenzen

https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/

Konstante

```
#include<iostream>
using namespace std;

class Test {
    const int t;
public:
    Test(int t):t(t) {}  //Initializer list must be used
    int getT() { return t; }
};

int main() {
    Test t1(10);
    cout<<t1.getT();
    return 0;
}
```

Referenz

```
class Test {
    int &t;
public:
    Test(int &t):t(t) {}  //Initializer list must be used
    int getT() { return t; }
};
```

1.2.5 Destruktor

```
class THuman:public TBeing {
   public:
    THuman(int nr):TBeing(nr){}          // Aufruf des Konstruktors Basisklasse
    ~THuman(){}
      int getLegs(){return legs;}        // Zugriff auf legs wegen protected
      void speak(){};
};
```

1.2.6 Polymorphie

Unterschiedliche Objekte haben die **gleiche Methoden-Namen**, der Compiler erkennt am Datentyp zur Übersetzungszeit, welche Funktion aufgerufen werden muss.

Will man unterschiedliche Objekte über einen gemeinsamen Vorfahren verwalten, wie soll der Compiler erkennen, dass er nicht die Methode der Basisklasse aufrufen soll?

Im folgenden Beispiel gibt es Polymorphie: sowohl Mensch als auch Hund kennen die Methode „springen()“. Springen bedeutet aber beim Mensch etwas anderes als beim Hund (2 Beine, 4 Beine).

Solange man die polymorphen Methoden mit dem zugehörigen Objekt verknüpft gibt es kein Problem. (z.B: dog.run(), being.run())

Soll aber der Compiler zur Laufzeit für ein Objekt das in ein Objekt der Basisklasse gespeichert ist die richtige Methode aufrufen, so muss dies über das Schlüsselwort **virtual** dem Compiler mitgeteilt werden:

Virtuelle Funktionen

Das Schlüsselwort virtual ist nötig, wenn ein Array von Objekten der Basisklasse existieren, aber die unterschiedlichen Funktionen der abgeleiteten Klassen aufgerufen werden sollen.

```
      #include <avr/io.h>
class TBeing {
      public:
      virtual void jump(){};
      void run(){}
};
class THuman:public TBeing {
      public:
      virtual void jump(){};
      void run(){}
};

class TDog:public TBeing {
      public:
      virtual void jump(){};
      void run(){}
};
```

```
int main(){
      TBeing b;
      TDog dog;
      THuman fred;
      b.run();          //TBeing::run()
      fred.run();       //THuman::run()

      TBeing* b1 = &fred;
      b1->jump();       //THuman::jump()

      TBeing beings1[3] = {b,dog,fred};
      beings1[0].run(); //TBeing.run()
      beings1[0].jump();//TBeing.jump()
      beings1[1].run(); //TBeing.run()          calls wrong Method
      beings1[1].jump();//TBeing.jump()         calls wrong Method

      TBeing* beings2[3] = {&b,&dog,&fred};     //Array of Addresses
      beings2[0]->run(); //TBeing.run()         // -> instead of .
      beings2[0]->jump();//TBeing.jump()
      beings2[1]->run(); //TBeing.run()         calls wrong Method
      beings2[1]->jump();//TDog.jump()          !!! Virtual Method !!!
      }
```

1.3 Kontrollfragen

Kontrollfragen und Begriffe

1.4 Anhang

1.4.1 Arduino Library Wire.h

Library Wire.h für I²C / TWI Schnittstelle.

TWI = I²C

Kleine Unterschiede: z.B: für TWI keine 10bit Adressierung/High speed mode

https://www.arduino.cc/en/Reference/Wire

- begin()
- requestFrom()
- beginTransmission()
- endTransmission()
- write()
- available()
- read()
- SetClock()
- onReceive()
- onRequest()