Objectgeoriënteerd programmeren

een korte inleiding met C++

Roland Stiglmayr (Duitsland)

Wilt u zelf krachtige objectgeoriënteerde programma's schrijven? Dan moet u zich eerst vertrouwd maken met de voordelen van OOP ten opzichte van procedureel programmeren. Laten we eerst wat theorie behandelen, om dan met enkele praktische voorbeelden verder te gaan.

Objectgeoriënteerd programmeren (OOP) bestaat al meer dan 40 jaar. Enkele van de vele voordelen ervan zijn de superieure kwaliteitsborgingsprocedures, het eenvoudige software-onderhoud en de uitstekende structurering die teamwerk bij programma-ontwikkeling vereenvoudigt. De afgelopen jaren heeft OOP ook zijn weg gevonden naar embedded programmeren. De Arduino IDE ondersteunt ook het gebruik ervan; het lijkt daarom een goed moment om het onderwerp wat nader te bekijken. Dit artikel kan de belangrijkste kenmerken van OOP slechts kort aanstippen, maar zal hopelijk een goed kennisfundament creëren waarop u uw eigen objectgeoriënteerde programma's kunt opbouwen.

Als het gaat om software-ontwikkeling, kan men in principe twee benaderingen gebruiken. De traditionele methode maakt gebruik van *procedureel programmeren*, in tegenstelling tot *objectgeoriënteerd programmeren*. Procedureel programmeren is wat we altijd hebben gedaan (bijvoorbeeld bij programmeren in assembler of in standaard C). Hierbij wordt het programma opgesplitst in kleine modules (dat wil zeggen procedures of functies) en sequentieel uitgevoerd. De modules communiceren met behulp van gemeen-

Listing 1. Klasse-declaratie van Virt_M (les_1).

```
class Virt_M
  {
    private:                                  //the following members are only
                                              //accessible within the class
      float voltage;                          //virtual voltage/ V
      float current;                          //virtual current/ A
      float p_result;                         //calculated power/ W
      float r_result;                         //calculated resistance/ Ohm.

    public:                                   //the following members are also
                                              //accessible from outside the class
      String s= ("public: Demo attribute s, type String");  //for demo only

  //  declaration of the constructor, its method is called when an object
  //  of this class is created

      Virt_M (float v, float c);              //declaration of the constructor

  //  declaration of the methods/ functions of the class Virt_M

      float get_P ();                         //method reads power
      float get_R ();                         //method reads resistance
      float get_Voltage ();                   //method reads voltage
      float get_Current ();                   //method reads current
      void prep_Meas (float v, float c);      //method simulates the measurement

    private:                                  //only accessible within the class

      void set_Voltage (float v);             //interface method for writing voltage
      void set_Current (float c);             //interface method for writing current

  };                                          //end of the class declaration
```

schappelijke, vaak globaal gedeclareerde data en gebruiken deze om resultaten van hun acties uit te wisselen. OOP daarentegen kapselt de gegevens en de functies die deze gebruiken in objecten in. Toegang tot gegevens en functies vindt plaats via hiervoor bestemde functies, ook wel interfaces genoemd. Inkapseling beschermt de data tegen ongeoorloofde toegang.
Wat zijn nu de kenmerken van OOP? Bij OOP worden functionele eenheden gecreëerd met logisch verwante leden zoals data en functies. De functies worden hier methoden genoemd. In het geval van een datalogger bijvoorbeeld zouden de data-acquisitie-eenheid met zijn vele meetkanalen, de verwerkingseenheid en de gebruikersinterface elk als een functionele eenheid kunnen worden beschouwd. Een functionele eenheid kan de eigenschappen van andere functionele eenheden erven, zodat uit deze basiseenheden een modulaire structuur kan worden opgebouwd. Bescherming van de data en methoden wordt bereikt door het principe van inkapseling.
We kunnen zo'n functionele eenheid beschouwen als een apart gegevenstype: een *klasse*. De klasse vertegenwoordigt een soort blauwdruk waaruit de variabelen worden gecreëerd. Die variabelen worden objecten of instanties van de betreffende klasse genoemd. Om misverstanden te voorkomen, moet u zich terdege bewust zijn van het verschil tussen een klasse en een object! Vanuit een klasse kan een willekeurig aantal objecten worden gemaakt. Van elke persoon die in een bedrijf werkt, zou zouden we kunnen zeggen dat hij een object voorstelt, waarbij elk van deze objecten is gemaakt vanuit een en dezelfde klasse.
Genoeg theorie, tijd voor de praktijk! Zoals altijd vormen kant-en-klare voorbeelden waar u aan kunt sleutelen en die u kunt aanpassen om uw eigen ideeën uit te proberen, de beste manier om te beginnen. En wanneer dit binnen een vertrouwde geïntegreerde ontwikkelomgeving (IDE) gebeurt, spaart dat tijd en is de leercurve minder steil. Om deze reden hebben we de populaire gratis Arduino IDE gebruikt om de hier gebruikte voorbeelden uit te voeren. De geïntegreerde compiler is een volwaardige C++ compiler en daardoor bij uitstek geschikt voor objectgeoriënteerd programmeren. Om de sketches in een Arduino-board te laden, wordt dit board aangesloten op de USB-poort van de computer. Onze voorbeeldprogramma's (in de Arduino-wereld worden die 'sketches' genoemd) zijn georganiseerd als zeven 'lessen' die kunnen worden gedownload van de projectpagina [1] bij dit artikel. Nadat ze zijn gedownload, kunnen ze worden uitgepakt en naar de map Arduino IDE Sketch worden gekopieerd. Als dat eenmaal is gebeurd, zijn de voorbereidingen voltooid. De voorbeelden zijn allemaal gebaseerd op het concept van een virtueel meetapparaat; er is geen hardware bij betrokken. We illustreren hiermee alleen de processen die betrokken zijn bij het overdragen van informatie. De voorbeeld-sketches zijn niet geschikt voor een 'echt' hardwarematig meetapparaat, maar ze demonstreren op aanschouwelijke wijze de basisfuncties van OOP aan de hand van een taak die gemakkelijk te begrijpen is.

De eerste klasse

Om te beginnen zullen we kennismaken met het creëren van een klasse, de betekenis van de toegangsspecificatie, de basisprincipes van inkapseling en het creëren van instanties. Hiertoe moeten we de sketch van *les_1* in de IDE laden, compileren en vervolgens naar het board overdragen. Voordat we de seriële monitor van de Arduino IDE gebruiken om de uitvoer te bekijken die door het programma wordt gegenereerd, werpen we eerst een blik op de broncode van het programma in **listing 1**.

Op de eerste regel zien we de descriptor-klasse met de naam `Virt_M`. Deze vertelt de compiler dat een klasse met de naam `Virt_M` moet worden gegenereerd. In de volgende regels kunnen we declaraties zien van de interne data (of attributen) die in de klasse worden gebruikt, en de methoden van de klasse. Voor toegangscontrole worden de leden van de klasse, de attributen en de methoden van de klasse voorzien van `private` en `public` toegangsspecificaties. Leden met `private` toegang zijn alleen toegankelijk vanuit de klasse. Alle leden met `public` toegang zijn toegankelijk van buiten de klasse. We zullen later zien wat dit inhoudt.
De eerste methode, die dezelfde naam heeft als de klasse en die geen waarde retourneert, is de constructor. De constructor wordt altijd aangeroepen wanneer een object van het type van deze klasse wordt gecreëerd. Aan de constructor worden meestal waarden doorgegeven om de objectattributen te initialiseren.
De andere methoden in het kort: de methoden die beginnen met `get_` staan toe dat de `private` attributen van de klasse worden gelezen. De methoden die beginnen met `prep_` worden gebruikt om deze attributen in te stellen. Deze methoden zijn `public` en brengen daardoor interfaces tot stand met de private attributen. De `set_` methoden die zijn opgegeven als `private` kunnen alleen binnen de klasse worden gebruikt. `String s` wordt gebruikt om de specificatie voor `public` toegang te demonstreren. De sluit-accolade gevolgd door een puntkomma geeft het einde van de declaratie van de klasse aan.
Hierna volgen de definities van de methoden die hierboven zijn gedeclareerd. Dit is identiek aan de definities van functies in standaard C, afgezien van de gebruikte syntaxis. Die luidt hier:

```
class name::method name(parameterst)
```

De verwijzing naar de klasse wordt tot stand gebracht met de *scope reference* operator, een dubbele dubbele punt.
Onze eerste klasse is nu klaar; van hieruit kunnen we objecten maken. Net als bij de functiedeclaratie bestaat de naam van de klasse uit het type gevolgd door de naam van het object en (indien overeengekomen) een lijst met waarden die moeten worden overgedragen (**listing 2**). Bij het maken van een object, ook wel instantiëren genoemd, wordt de constructormethode van de klasse aangeroepen en worden de attributen van het betreffende object gemaakt. Het is erg belangrijk om te beseffen dat elk object zijn eigen, individuele datagedeelte heeft, namelijk de attributen.
We hebben nog niet gezegd wat een object van deze klasse doet. In dit voorbeeld simuleert het de acquisitie van spanning- en stroom-meetwaarden; deze verzonnen waarden worden samen met de resulterende waarden voor weerstand en vermogen beschikbaar gesteld voor de corresponderende methoden. Een `public` methode wordt als volgt aangeroepen:

```
Object_name.method_name ();
```

Attributen die met `private` worden beschermd, zijn alleen toegankelijk met de methoden die daarvoor bestemd zijn. Als u probeert ze rechtstreeks te benaderen, retourneert de compiler een foutmelding. Dit kan worden gedemonstreerd door de betreffende commentaar-slashes (`//`) in de setup-routine weg te halen (**listing 3**).
Als ander voorbeeld van toegangscontrole hebben we `String s` opzettelijk als `public` gedeclareerd. Attributen die op deze manier zijn gedeclareerd, zijn rechtstreeks toegankelijk met `Objectname.`

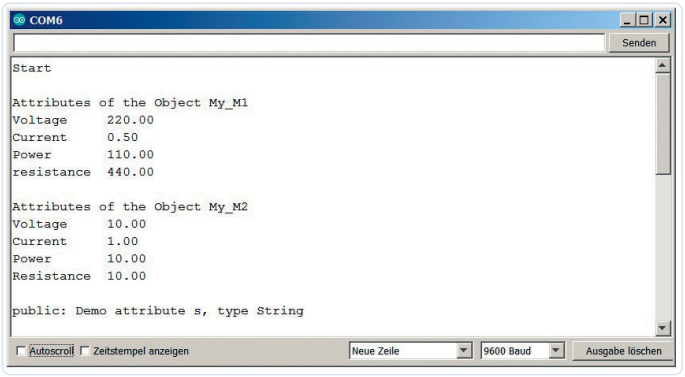
Listing 2. Creëren van de instanties (les_1).

```
// creating instances/ objects of type
// Virt_M. preset the attributes start values
//-----------------------------------------

   Virt_M My_M1 (220.0,0.5);      //creates the object/ instance My_M1 by
                                  //calling the constructor of Virt_M

   Virt_M My_M2 (10.0,1.0);       //creates the object/ instance My_M2 by
                                  //calling the constructor of Virt_M
```

Listing 3. Test van de toegangscontrole (les_1).

```
//  for testing the access control delete the comment instruction "//"
//  ------------------------------------------------------------------

//  My_M1.voltage= 0;              //error > attribute is private
//  My_M1.set_Voltage(0);          //error > method is private
   String str= (My_M1.s);          //permitted > attribute s is public
   Serial.println (My_M1.s);
```

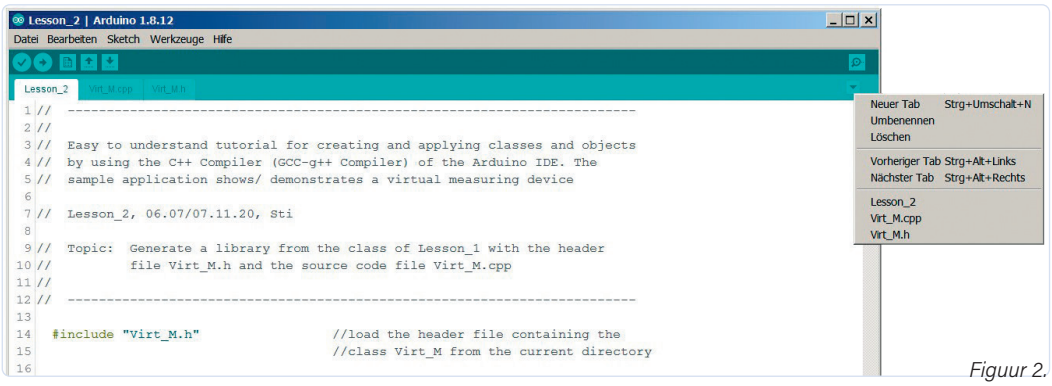
`Attributename`. Maar let op! Het object beschermt `public` leden niet tegen onjuist gebruik door het gebruikersprogramma.
Het hoofdprogramma verandert de gelezen waarden met behulp van de methode `prep_Meas`, die de interne `set` methoden aanroept. De programmavoorbeelden bevatten nuttig commentaar en veel informatie, die moeten bijdragen aan een beter begrip. Na het starten van de applicatie voert de setup-routine de attributen van de objecten uit zodat we die kunnen bekijken met behulp van de seriële monitor (**figuur 1**).



Figuur 1. Uitvoer van de setup-routine van les_1.

Een klasse als programmabibliotheek

Het concept van bibliotheken is niet nieuw in de wereld van software-ontwikkeling. U hebt ze al gebruikt, als u enige ervaring hebt met coderen in standaard C. Ze worden nog intensiever gebruikt in OOP. De reden hiervoor is dat de code-ontwikkelaar de details van de broncode van een klasse moet kennen om er gebruik van te kunnen maken. Het gebruik van bibliotheken ondersteunt de herbruikbaarheid van programma-onderdelen en bevordert daarmee het concept van modulariteit in het programma-ontwerp. Een voorbeeld hiervan wordt gegeven in *les_2*, waar de klasse uit de eerste sketch wordt gebruikt als programmabibliotheek. Een bibliotheekitem bestaat uit twee bestanden. Het headerbestand (.h) bevat het declaratiegedeelte van de klasse en het brontekst-bestand (.cpp) de methodedefinitie van de klasse. Hoewel het niet strikt noodzakelijk is, is het handig om voor beide bestanden dezelfde naam te gebruiken. Om onze `Virt_M` klasse als een bibliotheek weer te geven, hoeft alleen het declaratiedeel naar het headerbestand te worden overgebracht en het definitiedeel naar het brontekstbestand. Tijdens de ontwikkeling wordt aanbevolen om de bibliotheekbestanden op te slaan in dezelfde directory als het gebruikersprogramma. Het gebruik van de tab-functie in de Arduino IDE helpt ons hierbij, maar daarover later meer.
Het is gebruikelijk om aan het begin van het header-bestand te controleren of het al door de compiler is opgenomen. Die situatie kan zich voordoen als een ander header-bestand al onze header heeft ingesloten. Als dat het geval is, worden leden met dezelfde naam meerdere keren gedeclareerd. De syntax voor de preprocessor om dit te testen is:



Figuur 2. De Arduino tab-functie (les_2).

```
#ifndef  Virt_M_h  // the symbol Virt_M_h is used to
   test if the header file was already integrated
#define Virt_M_h   // if the symbol does not yet exist,
   then define the symbol
  class ....       // now declare the class
    {   ....  };   // end of declaration
#endif             // finish conditional compiling
```

De `#include "Virt_M.h"` instructie in het brontekstbestand wordt gebruikt om het headerbestand op te nemen. In ons voorbeeld is `Arduino.h` ook opgenomen. Dit is nodig omdat de compiler op dit moment de standaardbibliotheken nog niet kent. Daarmee is onze bibliotheek klaar. Om nu de `Virt_M` klasse in een gebruikersprogramma te gebruiken, kunnen we `#include "Virt_M.h"` gebruiken om het header-bestand te laden.
Arduino-bibliotheken bieden vaak al een instantie van hun klassebibliotheek. In de Wire-bibliotheek is *Wire* bijvoorbeeld een object van de klasse *TwoWire*. Het object wordt opgenomen met `#include <Wire.h>`.
Nadat de ontwikkeling is voltooid, kunnen we onze bibliotheekbestanden in hun eigen directory opslaan. Hiervoor maakt Arduino de bibliotheekdirectory aan in de sketchbook-directory. We kunnen daar een andere map maken waarin we onze bestanden zullen opslaan. Dit werkt ook via de IDE met *Sketch -> Include library -> Add .ZIPlibrary*. Zoals al opgemerkt, is de tab-functie van de Arduino IDE erg handig bij het schrijven van meerdere bestanden die bij een project horen. Deze functie wordt geactiveerd via de kleine driehoek in de rechterbovenhoek van het venster (**figuur 2**).

Overerving – een kernconcept van OOP

Als u zich eenmaal in OOP begint te verdiepen, duurt het niet lang voordat u het idee van overerving tegenkomt. Laten we eens kijken naar de voordelen van dit concept. Overerving betekent het beschikbaar stellen van de attributen en methoden van een basisklasse aan een overervende klasse. De overervende klasse kan een onderliggende klasse, afgeleide klasse of subklasse worden genoemd. De basisklasse wordt ook wel de bovenliggende klasse of superklasse genoemd. De afgeleide klasse voegt extra eigenschappen toe aan die welke worden geërfd en die ook verder kunnen worden overerfd. Dit betekent dat veel verschillende afgeleide klassen kunnen voortkomen uit een overervende klasse, zodat de resulterende structuur niet noodzakelijk lineair is, maar eventueel vertakt. Een relatie die bestaat uit vele takken!
Les_3 laat zien hoe we overerving kunnen gebruiken voor onze doeleinden. Om het voorbeeld wat zinvoller te maken, gebruiken we multiplex-adressen om de verschillende meetkanalen te selecteren en deze te koppelen aan de aangemaakte objecten. `Virt_M` is de basisklasse. Hier kunt u zien dat er drie constructors zijn met een verschillend aantal parameters. C++ laat in het algemeen dit zogenaamde 'overladen' van functiedefinities toe (dat wil zeggen functies die dezelfde naam gebruiken met verschillende nummers en/of verschillende parametertypes). In OOP staat dit bekend als *polymorfisme*. Merk op dat de sleutelwoord-toegangsspecificatie is ingesteld op `protected`. Dit stelt de toegankelijkheid van methoden, klassen en andere leden in voor de overervende klasse. In ons voorbeeld geeft het aan dat alle overervende klassen toegang hebben tot de `set_Mux` methode.

Om te tonen dat dit is toegestaan, is de methode gedefinieerd in het declaratiegedeelte (dit heeft alleen werkelijk zin voor zeer korte functies). De eerste afgeleide klasse is `Volt_M`, die de klasse `Virt_M` erft. De syntax hiervoor is:

```
class Volt_M : public Virt_M   // class Volt_M
     // inherits the class Virt_M
```

Het keyword `public` zorgt ervoor dat de toegangscontrole tot de leden van `Virt_M` behouden blijft. `Volt_M` levert het `sV` attribuut, de constructor en de `get_Unit` methode aan zijn overgeërfde leden. De declaratie van de constructor introduceert niets nieuws, maar de definitie wel. Hier wordt de constructor 2 van `Virt_M` aangeroepen. Tijdens instantiatie worden de parameters die aan `Volt_M` worden doorgegeven, doorgegeven aan de `Virt_M` klasse, de startwaarde `ini_m` rechtstreeks via de constructor `Virt_M`, het multiplex-adres via de `set_Mux` methode van `Volt_M`. De definitie van de constructors is:

```
Volt_M::Volt_M (byte v_mux, float ini_m):Virt_M
   (ini_m)  //set ini_m via constructor of of class
     // Virt_M
   { set_Mux (v_mux); }
//v_mux via set method
```

De klasse `Amp_M` is ook een erfenis van `Virt_M`. Deze heeft een vergelijkbare structuur als `Volt_M` met het verschil dat de constructor de constructor 3 aanroept vanuit `Virt_M`. De volledige overdracht van waarden naar `Virt_M` wordt uitgevoerd zonder enige aanvullende methode. Het is vermeldenswaard dat een overerving een overgeërfde methode kan opheffen.
De klassen zijn nu volledig geformuleerd, zodat we er exemplaren van kunnen maken. De objecten `M1`, `M2` en `M3` van klassetype `Virt_M` tonen de aanroep van de overladen constructor. `My_Volt_M` en `My_Amp_M` zijn objecten van de overgeërfde klassen `Volt_M` en `Amp_M`. Na het starten van het programma kan het gedrag van onze objecten worden gevolgd en begrepen met behulp van de seriële monitor. Ongeldige toegangen zijn opzettelijk in de sketch ingebouwd, maar worden in het voorbeeld in eerste instantie weggecommentarieerd. Als we de commentaartekens `//` verwijderen en het geheel nogmaals compileren, kunnen we de foutmeldingen die door de compiler worden gegenereerd bestuderen.
We moeten ons ervan bewust zijn dat de overervende objecten de overgeërfde objecten gebruiken alsof ze een deel van zichzelf zijn. Elk object heeft echter zijn eigen individuele dataset. De relatie is bijna alsof de code van `Virt_M` is gekopieerd en geplakt in `Volt_M` en `Amp_M`. Het grootste nadeel van dat soort relatie (afgezien van het extra werk en de omvangrijkere code) zou zijn wanneer er wijzigingen worden aangebracht in `Virt_M`. Alle andere klassen die `Virt_M` bevatten, moeten dan ook worden gewijzigd.

Een klasse kan ook een vriend zijn

In het laatste voorbeeld zagen we dat het vanwege inkapseling niet mogelijk is om het multiplex-adres van een object te wijzigen. Vooral bij het programmeren (vrijwel) op hardwareniveau is het echter vaak wenselijk om over veel toegangsopties te beschikken. Als we bijvoorbeeld alle multiplex-adressen opeenvolgend willen adresseren tijdens het debuggen, moeten we voor elk adres een object maken. Een andere mogelijkheid zou zijn om het object na de oproep te vernietigen en het vervolgens opnieuw te maken. We zouden de basisklasse natuurlijk vanaf het begin zo kunnen formuleren dat de toegang openbaar is. Dit zou echter betekenen dat

Listing 4. Klasse-declaratie van Virt_M met Debug_M als 'Friend' klasse (les_4)

```
class Virt_M
  {
    friend class Debug_M;                //the class Debug_M gets
                                         //access to the private members
                                         //of Virt_M

    private:                             //internal members of the class
      float value=0;                     //virtual measurement value, set to 0
      byte mux_adr= MUX_INVALID;         //mux address for simulation

    public:                              //members also accessible from outside
      Virt_M (byte mux, float ini_m);    //constructor with initial values
      void set_Value (float val);        //method writes the virtual measurement
      float get_Value ();                //method reads the virtual measurement
      byte get_Mux ();                   //method reads mux address
      String s="public: Demostring";     //for demo only

  };                                     //end of the class declaration
```

Listing 5. Declaratie van het structuur-datatype t_result (les_5).

```
struct t_result     //type for returning data
  {
    byte mux;       //mux adress
    String s;       //a string
    float val;      //measurement value of type float
  };
```

Listing 6. Attribuut-declaratie van de Dev_M klasse (les_5).

```
class Dev_M
  {
    private:
      Volt_M *ptr_v;                     //pointer to an object of type Volt_M
      Amp_M *ptr_a;                      //pointer to an object of type Amp_M
      t_result result;                   //for returning data
      const String sP=( "Power/ W   : ");
      const String sR= ("Resist/ Ohm: ");
```

een van de voordelen van OOP wordt opgegeven, namelijk dat de gegevens tot op zekere hoogte beschermd zijn. *Les_4* toont een nette oplossing voor dit probleem. We overerven de basisklasse `Virt_M` naar een nieuwe klasse met de naam `Debug_M` en promoveren deze tot *friend* in de basisklasse. De functie van `Debug_M` is identiek aan die van `Volt_M` met de uitzondering dat deze ook toegang heeft tot de private leden van `Virt_M`. Om het multiplex-adres te wijzigen, kunnen we de `set` methode in `Debug_M` gebruiken die als public is gedeclareerd. **Listing 4** toont de declaratie van een `friend` klasse. De klasse `Virt_M` bevat geen methode om het multiplexadres in te stellen. In plaats daarvan krijgt de constructor twee waarden voor het initialiseren van de klassekenmerken. Een van deze waarden is het multiplexadres. De parameters worden doorgegeven wanneer de basisklasse en de subklassen worden geïnstantieerd.

Opnieuw toont de uitvoer van de `setup` routine de resultaten van onze inspanningen. `Friend` klassen worden zelden gebruikt, hoewel ze het voordeel bieden dat ze gebruik maken van bestaande methoden en de gewenste functionaliteit bieden zonder het beveiligingsconcept van inkapseling bij correct gebruik in gevaar te brengen.

Een klasse die toegang heeft tot een methode van een andere klasse

In de laatste voorbeelden hebben we vermogens- en weerstandsberekeningen uitgevoerd in het hoofdprogramma. We lezen de attributen van `Volt_M` en `Amp_M` via hun methoden. Nu gaan we dit doen in een aparte klasse. *Les_5* definieert de nieuwe klasse `Dev_M`, die toegang heeft tot methoden van de objecten `Volt_M` en `Amp_M`. Hiervoor worden, wanneer een instantie van `Dev_M` wordt

Listing 7. De constructormethode van Dev_M slaat de pointer op (les_5).

```
    //  the constructor passes the pointers to the objects of Volt_M, Amp_M
    //  ---------------------------------------------------------------------

          Dev_M::Dev_M (Volt_M *ptrv, Amp_M *ptra)
            {
               ptr_v= ptrv;              //pointer to Volt_M
               ptr_a= ptra;              //pointer to Amp_M
            }
```

Listing 8. Definitie van de get_Powr-methode van Dev_M (les_5).

```
t_result Dev_M::get_Powr ()     //calculate power
  {
    result.s= sP;               //name, unit
    result.val= ptr_v -> get_Value () * ptr_a -> get_Value ();
    return result;              //return via type t_result
  }
```

Listing 9. Instantiëren van de objecten, pointers naar de objecten overdragen (les_5).

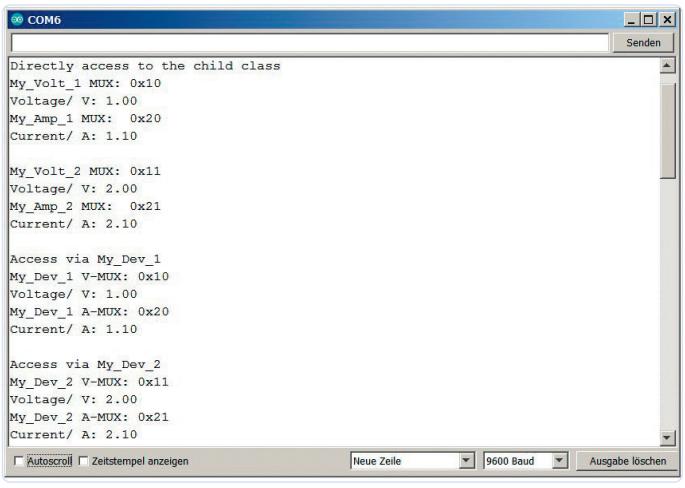
```
  Volt_M My_Volt_1 (V_MUX0, 1.00);          //object of type Volt_M
  Volt_M My_Volt_2 (V_MUX1, 2.00);          //object of type Volt_M
  Amp_M My_Amp_1 (A_MUX0, 1.10);            //object of type Amp_M
  Amp_M My_Amp_2 (A_MUX1, 2.10);            //object of type Amp_M
  Dev_M My_Dev_1 (&My_Volt_1, &My_Amp_1);   //object of type Dev_M, passing pointers
  Dev_M My_Dev_2 (&My_Volt_2, &My_Amp_2);   //object of type Dev_M, passing pointers
```

gemaakt, verwijzingen naar de aan te roepen objecten door de constructor doorgegeven aan Dev_M.
Eerst maken we echter een gestructureerd datatype met behulp van `struct` dat uit verschillende elementen bestaat die de resultaten bevatten. Het datatype heeft de naam `t_result` (**listing 5**). Daarop volgt de declaratie van de `Dev_M` klasse. Hier worden de pointers `ptr_v` en `ptr_a` toegewezen die worden gebruikt in de klassen `Volt_M` en `Amp_M`. De `*` operator geeft aan dat de erop volgende naam moet worden geïnterpreteerd als een pointer (**listing 6**). De constructormethode slaat de doorgegeven pointers op bij de attributen (**listing 7**).
De `Dev_M` methode kan nu de pointers gebruiken om toegang te krijgen tot de externe objecten. Dat wordt gedaan met behulp van de instructie `pointername -> methodname`. De `->` operator geeft toegang tot een methode met behulp van een pointer in plaats van de `.` (punt) operator (**listing 8**).

Om een instantie van `Dev_M` te kunnen maken, is het nodig om vooraf nieuwe instanties van de indexklassen te initialiseren. Deze moeten van het type `Volt_M` en `Amp_M` zijn. Bij het instantiëren van `Dev_M` voeren we de verwijzingen naar deze objecten in de constructor-parameterlijst in. Hiervoor gebruiken we de operator `&` (**listing 9**). De uitvoer van de `Setup` routine in **figuur 3** in onze applicatie laat zien dat de methode-aanroep van de `My_Dev_1` en `My_Dev_2` objecten alle relevante resultaten opleveren.

Meervoudige overerving

In het vorige voorbeeld moesten we eerst twee afzonderlijke instanties maken voordat we de methoden van `Dev_M` konden gebruiken. Er zijn toepassingen waar dit zeker zinvol is, maar voor ons is het vooral een introductie in het gebruik van pointers naar objecten. Sommigen vragen zich misschien af of basis- en afgeleide klassen simpelweg kunnen worden geërfd door een andere klasse. Dit is



Figuur 3. Uitvoer van de setup-routine van les_5.

Listing 10. Expliciete verwijzing naar een lid in geval van dubbelzinnigheid (les_6).

```
t_result Dev_M::get_Volt ()              //read the current voltage
  {
    result.s= Volt_M::get_Unit ();       //call the Volt_M method
    result.val= Volt_M::get_Value ();    //call the Virt_M method
    return result;
  }
```

Listing 11. Ambiguïteit van de get_Mux-methode (les_6).

```
  Serial.print ("My_Dev_1, Volt MUX: 0x"),
  Serial. println (My_Dev_1.Volt_M::get_Mux(), HEX);   //method of base class
  Serial.print (My_Dev_1.get_Volt().s);
  Serial.println (My_Dev_1.get_Volt().val);            //initial value of instantiation of My_Dev_1
  Serial.println ();
```

Listing 12. Een methode gebruiken om de dubbelzinnigheid op te lossen (les_6).

```
  Serial.print ("My_Dev_2, Volt MUX: 0x");
  Serial.println (My_Dev_2.get_VMux(),HEX);            //method of the inherited class
  Serial.print (My_Dev_2.get_Volt().s);
  Serial.println (My_Dev_2.get_Volt().val);
  Serial.println ();
```

precies wat is geïmplementeerd in *les_6*: meervoudige overerving. De twee afgeleide klassen `Volt_M` en `Amp_M` worden overgeërfd door de basisklasse `Dev_M`. Omdat de afgeleide klassen `Virt_M` bevatten, heeft `Dev_M` toegang tot leden van alle klassen.
Om eenvoudig de complexe resultaten van de `Dev_M` methoden te retourneren, declareren we nogmaals het structuurgegevenstype `t_result`. Wanneer de klasse is gemaakt, worden de overgeërfde klassen weergegeven (gescheiden door komma's) na de `:` operator:

```
class Dev_M: public Volt_M, public Amp_M
```

Met de volgende constructor-declaratie worden de parameters die moeten worden doorgegeven aan de overervende klassen, toegewezen in een parameterlijst:

```
Dev_M (byte v_x, float i_v, byte a_x, float i_a);
   // the constructor of class Dev_M
```

De definitie van de constructormethode wijst vervolgens de parameters toe aan de overdrachtswaarden van de overervende klassen. Bij het instantiëren van `Dev_M` worden de constructormethoden `Volt_M` en `Amp_M` aangeroepen en worden de parameterattributen van deze klassen toegewezen.

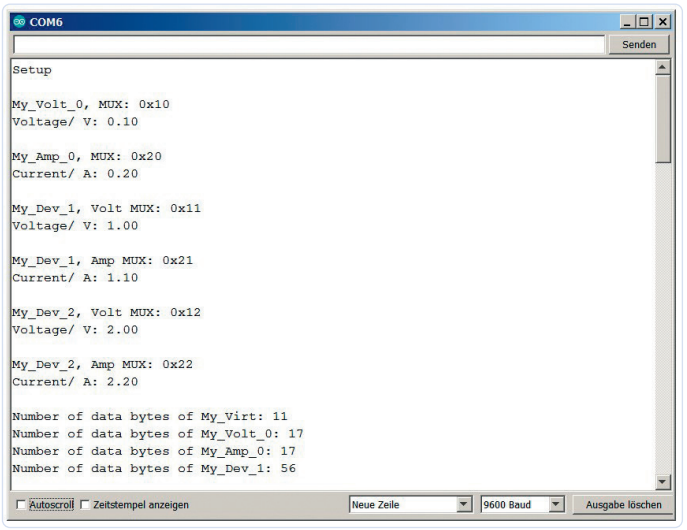
```
Dev_M::Dev_M (byte v_x, float i_v, byte a_x, float i_a):
Volt_M (v_x, i_v), Amp_M (a_x, i_a){}
```

Zoals eerder vermeld, hebben we toegang tot alle attributen en methoden van de overervende klassen via `Dev_M`, op voorwaarde dat de toegangsspecificatie dit toestaat. `Dev_M` biedt ons geschikte methoden om dit te doen. Er doet zich hier echter een probleem voor: aangezien de twee overgeërfde klassen zelf zijn afgeleid van de `Virt_M` klasse, hebben ze methoden en attributen met dezelfde naam. Dat leidt tot onduidelijkheden. Om dit op te lossen gebruiken we de scope resolution operator `::` om expliciet aan te geven welk lid van welke klasse we bedoelen (**listing 10**).
Onze klassen zijn nu gedefinieerd en we kunnen er objecten van instantiëren. In principe zijn de methoden van `Dev_M` volledig voldoende om een applicatie te schrijven. Voor een beter begrip maken we ook meer instanties van de basisklasse en de afgeleide klassen. Omdat bij instantiatie aan elk object een ander multiplex-adres wordt toegewezen, kunnen we zien welk object momenteel wordt geadresseerd.
Nadat we de applicatie hebben gestart, toont de seriële monitor de resultaten van de methode-aanroepen. Het uitvoeren van de `Setup` routine is interessant. De attributen van de basisklasse worden op verschillende manieren gelezen. Via het multiplexadres van het `My_Dev_1` object zal de methode van de basisklasse worden gebruikt. Zoals we bij de methodedefinities van `Dev_M` hebben opgemerkt, doet zich hier ook het probleem van ambiguïteit voor. Om het multiplex-adres te lezen, moeten we expliciet specificeren of we het adres van de spanning of van de stroom bedoelen. De methode wordt in beide gevallen `get_Mux` genoemd (**listing 11**). We lossen het probleem op door de klasse-referentie te specificeren.
Het is natuurlijk veel duidelijker en uiteindelijk eenvoudiger om te voorzien in een eigen methode in de afgeleide klassen `Volt_M` en `Amp_M` (**listing 12**). In het voorbeeld heten deze methoden respectievelijk `get_VMux` en `get_AMux`. Deze worden gebruikt bij het lezen van de attributen van `My_Dev_2`.
De `sizeof` operator wordt gebruikt om het aantal bytes in de datavelden van ons object te bepalen. De basisklasse reserveert 11 bytes, terwijl klassen die hiervan afgeleid zijn 17 bytes gebruiken. Omdat de basisklasse in elke afgeleide klasse is opgenomen, nemen de afgeleide klassen zelf elk 6 bytes in beslag. `Dev_M` erft twee keer 17 bytes en heeft zelf 22 bytes nodig; in totaal zijn dat 56 databytes (**figuur 4**).

Iets lekkers voor Arduino-fans

Ik weet zeker dat er Arduino-fans zijn die regelmatig een van de Arduino-methoden in hun programma willen integreren. Misschien heeft u een eigen display ontwikkeld en wilt u daar naar toe kunnen schrijven met de werkelijk krachtige `print` methode . Zoals reeds gesuggereerd, kan dit worden bereikt door een klasse te maken die erft van de klasse `Print`. *Les_7* is een sketch die laat zien hoe dat gedaan kan worden. Onze klasse `My_Print_C` erft alle methoden van `Print` en overschrijft de oorspronkelijke `write` methode, die normaal gesproken weer te geven data naar specifieke hardware overdraagt. De `write` methode ontvangt een enkel teken van de `print(ln)` methode als argument en stuurt die vervolgens naar het display via een hardware-interface zoals de I²C-bus. Ons voorbeeld gebruikt geen hardware, dus de overgedragen tekens worden naar een string geschreven, waarvan de inhoud kan worden bekeken op de seriële monitor.

Omdat de `print(ln)` methoden overladen zijn, maakt het nauwelijks uit wat voor soort waarden we kunnen 'printen'. Dat is alles. Nu kunnen we `print` gebruiken om karakters naar ons eigen display te schrijven.



Figuur 4. Uitvoer van de setup-routine van les_6.

Klaar

Ik hoop dat dit artikel u een goede introductie heeft gegeven in de principes van objectgeoriënteerd programmeren. We hebben gekeken naar klasse-bibliotheken en hoe we deze in onze applicaties kunnen integreren. In het laatste voorbeeld hebben we getoond dat het overerven en wijzigen van externe klassen geen probleem is. Hoewel er nog veel te leren valt, zouden de basisprincipes voor het schrijven van uw eigen krachtige objectgeoriënteerde programma's nu allemaal aanwezig moeten zijn.

Ik hoop dat u overtuigd bent van de voordelen van OOP ten opzichte van procedureel programmeren. De concepten zijn in het begin misschien moeilijk te begrijpen, vooral als u vertrouwd bent met procedureel programmeren. Maar geen angst – het gezegde luidt niet voor niets 'oefening baart kunst'!

200563-03

Een bijdrage van

Idee en tekst: **Roland Stiglmayr**

Redactie: **Rolf Gerstendorf**
Layout: **Giel Dols**

Vragen of opmerkingen?
Hebt u vragen of opmerkingen naar aanleiding van dit artikel? Stuur een e-mail naar de auteur of naar de redactie van Elektor, via redactie@elektor.com.

WEBLINKS

[1] **Projectpagina bij dit artikel**: www.elektormagazine.nl/200563-03

[2] **C++ Programming**: https://en.wikibooks.org/wiki/Subject:C%2B%2B_programming_language