programmation orientée objet

Une brève introduction avec le C++

Roland Stiglmayr (Allemagne)

Vous voulez écrire de puissants programmes orientés objets ? Pour commencer, il vous faut familiariser avec les avantages de la programmation orientée objet par rapport à la programmation procédurale. D'abord un peu de théorie avant de passer aux exemples pratiques.

La programmation orientée objet (POO) existe depuis plus de 40 ans. Parmi ses nombreux avantages, citons ses procédures d'assurance qualité élaborées, une maintenance logicielle plus facile et une excellente structuration qui contribue à simplifier le travail en équipe dans le développement de logiciels. Ces dernières années, la POO a aussi trouvé son emploi dans le domaine du logiciel embarqué. L'EDI Arduino permet également son utilisation ; c'est le bon moment pour approfondir le sujet. Cet article ne donnera qu'un aperçu des principales caractéristiques de la POO, mais ce sera suffisant pour vous apporter une bonne base de connaissances et assez d'expérience pour vous inviter à écrire vos propres programmes orientés objet.

En matière de code informatique, il y a deux approches différentes : la programmation procédurale traditionnelle et la programmation

Listage 1. Déclaration de la classe Virt_M (leçon 1).

```
class Virt_M
  {
    private:                          //les membres suivants sont seulement
                                      //accessibles à l'intérieur de la classe
      float voltage;                  //tension virtuelle/ V
      float current;                  //courant virtuel/ A
      float p_result;                 //puissance calculée/ W
      float r_result;                 //résistance calculée/ Ohm.

    public:                           //les membres suivants sont également
                                      //accessibles depuis l'extérieur de la classe
      String s= ("public: Demo attribute s, type String");  //for demo only

  // Déclaration du constructeur, appelé quand un objet
  // de cette classe est créé

      Virt_M (float v, float c);      //déclaration du constructeur

  // Déclaration des méthodes/ fonctions de la classe Virt_M

      float get_P ();                    //méthode de lecture de la puissance
      float get_R ();                    //méthode de lecture de la résistance
      float get_Voltage ();              //mméthode de lecture de la tension
      float get_Current ();              //méthode de lecture du courant
      void prep_Meas (float v, float c);  //méthode de simulation de la mesure

    private:                          //seulement accessible de l'intérieur de la classe

      void set_Voltage (float v);     //méthode d'écriture de la tension
      void set_Current (float c);     //méthode d'écriture du courant

  };                                  //fin de la déclaration de la classe
```

orientée objet. La première est celle que nous avons toujours utilisée (par ex. pour la programmation en assembleur ou en C standard). Le programme est décomposé en petits modules (des procédures ou des fonctions) et exécuté de manière séquentielle. Les modules communiquent en utilisant des données communes, souvent déclarées globales, et s'en servent pour échanger les résultats de leurs traitements. En revanche, la POO enferme les données et les fonctions qui les utilisent dans des *objets* et en réserve l'accès à des fonctions spécifiques, l'*interface* de l'objet. Cette *encapsulation* protège les données contre tout accès inapproprié.

Avec la POO, les unités fonctionnelles sont constituées de membres logiquement liés entre eux, comme les données et les fonctions. Les fonctions sont appelées des *méthodes*. Prenez un enregistreur de données avec son unité d'acquisition de données équipée de nombreux canaux de mesure, son unité de traitement et son interface utilisateur. Chaque unité pourrait représenter une unité fonctionnelle. Une unité fonctionnelle peut hériter des propriétés d'autres unités fonctionnelles, de sorte qu'une structure modulaire peut être constituée à partir d'unités de base. La protection des données et des méthodes est assurée par le principe de l'encapsulation.

Une unité fonctionnelle peut être considérée comme un nouveau type de données appelé *classe*. La classe représente un « plan de construction » à partir duquel sont créées des entités appelées *objets* ou *instances* de cette classe. Attention à la différence entre la classe et l'objet ! Un nombre quelconque d'objets peut être *instancié* à partir d'une classe. Chaque personne travaillant dans une entreprise peut être considérée comme un objet créé à partir d'une seule et même classe.

Assez de théorie. Passons à la pratique ! Comme toujours, le plus simple est de commencer par des exemples tout faits que vous pouvez manipuler et modifier pour essayer vos propres idées. Le faire dans un environnement de développement intégré (EDI) familier fait gagner du temps et diminue la pente de la courbe d'apprentissage. Nous avons choisi l'EDI Arduino, populaire et gratuit, pour traiter les exemples proposés. Il intègre un compilateur C++ complet qui convient parfaitement à la POO. Nos exemples de programmes, appelés *croquis* dans le monde Arduino, répartis en sept « leçons », sont téléchargeables à partir de la page du projet [1] de cet article. Une fois téléchargés, il faut les décompresser et les copier dans le dossier *Sketch* de l'EDI Arduino ; l'environnement est alors prêt. Les exemples sont tous basés sur un appareil de mesure virtuel, sans réalité physique. Simples illustrations des processus impliqués dans le transfert d'informations, ils ne sont pas conçus pour gérer un appareil de mesure « réel » comprenant du matériel, mais pour présenter les éléments de base de la POO au moyen d'un logiciel facile à comprendre.

Une première classe

Pour commencer, nous apprendrons la création d'une classe, la signification du spécificateur d'accès, les bases de l'encapsulation et la création d'instances. Pour ce faire, il faut charger le croquis de la *leçon 1* dans l'IDE, le compiler et le télécharger sur la carte. Avant de visualiser la sortie produite par le programme sur le moniteur série de l'EDI Arduino, examinons d'abord son code source (**listage 1**).

Sur la première ligne, la déclaration `classe` est suivie du nom `Virt_M`. Cela indique au compilateur qu'une classe portant le nom `Virt_M` va être définie. Les lignes suivantes déclarent les variables internes (ou *attributs*) et les méthodes appartenant à la classe. Le contrôle d'accès est précisé en attribuant aux membres, aux attributs et aux méthodes de la classe l'un des spécificateurs d'accès `private` ou `public`. Les membres ayant un accès `private` ne sont accessibles qu'à l'intérieur de la classe. Tout membre ayant un accès `public` peut être accessible depuis l'extérieur de la classe. Nous verrons plus tard ce que cela signifie.

La première méthode, qui porte le même nom que la classe et ne retourne aucune valeur, est le *constructeur*. Le constructeur est toujours appelé lorsqu'un objet du type de cette classe est créé. Si des valeurs sont passées au constructeur, elles servent à initialiser les attributs de l'objet.

Parmi les autres méthodes, celles qui commencent par `get_` servent à lire les attributs privés de la classe ; celles qui commencent par `prep_` sont utilisées pour modifier ces attributs. Ces méthodes sont publiques et constituent l'interface de la classe. Les méthodes `set_` spécifiées comme `private` ne peuvent être appelées qu'au sein de la classe. La chaîne `string s` est utilisée pour démontrer le spécificateur d'accès `public`. L'accolade fermante suivie d'un point-virgule termine la déclaration de la classe.

Viennent ensuite les définitions des méthodes qui ont été déclarées ci-dessus. Elles sont identiques aux définitions des fonctions du C standard, sauf pour la syntaxe utilisée, qui est la suivante :

```
nom de la classe::nom de la méthode(liste des paramètres)
```

La référence à la classe est effectuée avec l'opérateur de résolution de portée, le double deux-points.

Notre première classe est maintenant prête à servir à créer des objets. Comme pour la déclaration de fonction, le nom de la classe est composé du nom de la classe suivi du nom de l'objet et (s'il y a lieu) d'une liste de valeurs initiales (**listage 2**). Lors de la création (ou instanciation) d'un objet, le constructeur de la classe est appelé et les attributs de cet objet sont créés et initialisés. Il faut bien avoir à l'esprit que chaque objet possède sa propre copie des attributs.

Quelle est l'activité d'un objet de cette classe ? Dans notre exemple, il simule l'acquisition de valeurs mesurées de tension et de courant et met ces valeurs à la disposition des méthodes correspondantes avec les valeurs de résistance et de puissance qui en sont déduites. L'appel d'une méthode publique est fait ainsi :

```
Nom_objet.nom_méthode();
```

Les attributs protégés par `private` ne sont accessibles qu'en utilisant les méthodes prévues à cet effet. Toute tentative d'accès direct est refusée par le compilateur avec un message d'erreur, comme on peut le vérifier en supprimant le marqueur de commentaire `//` au début d'une ligne de code du **listage 3**.

Comme autre exemple de contrôle d'accès, la chaîne `string s` a été délibérément déclarée `public`. Les attributs déclarés de cette manière sont accessibles directement par `nom_objet.nom_attribut`. Mais attention ! Ils ne bénéficient d'aucune protection contre une utilisation incorrecte par le programme utilisateur.

Le programme principal modifie les valeurs lues en utilisant la méthode `prep_Meas`, qui appelle les méthodes internes `set_`.

Les exemples de programmes contiennent des commentaires utiles et de nombreuses informations qui devraient contribuer à une meilleure compréhension. Après le démarrage de l'application, la routine de configuration affiche la liste des attributs des objets sur le moniteur série (**fig. 1**).

Listage 2. Création des instances (leçon 1).

```
// Création des instances / objets de type Virt_M
// Initialisation des attributs
//-----------------------------------------

   Virt_M My_M1 (220.0,0.5);  //crée l'object My_M1 en appelant
                              //le constructeur de Virt_M

   Virt_M My_M2 (10.0,1.0);   //crée l'object My_M2 en appelant
                              //le constructeur de Virt_M
```

Listage 3. Test du contrôle d'accès (leçon 1).

```
// Pour tester le contrôle d'accès, effacer l'instruction de commentaire "//"
//  ------------------------------------------------------------------

//  My_M1.voltage= 0;               //erreur > l'attribut est privé
//  My_M1.set_Voltage(0);           //erreur > la méthode est privée
  String str= (My_M1.s);            //permis > l'attribut s est public
  Serial.println (My_M1.s);
```

Une classe comme bibliothèque de programmes

Le concept de bibliothèque n'est pas nouveau dans le monde de la programmation informatique. Vous en avez sûrement déjà utilisé si vous avez une quelconque expérience du codage en C standard. Elles sont d'usage encore plus intensif en POO.

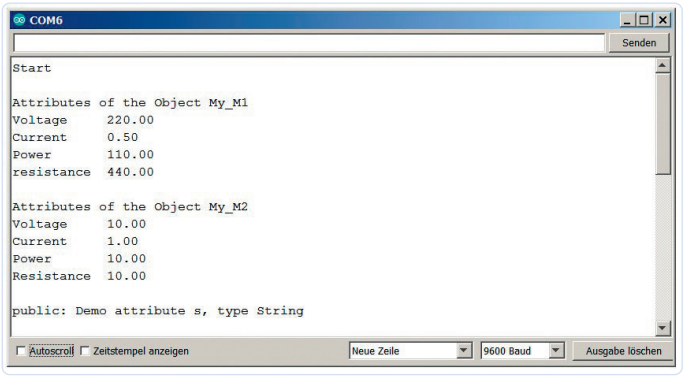


Figure 1. Sortie de la routine de configuration de la leçon 1.

La raison en est que le développeur a besoin de connaître les détails du code source d'une classe pour pouvoir l'utiliser. L'utilisation de bibliothèques encourage la réutilisation de parties de programme et promeut ainsi le concept de modularité dans la conception du programme.
Un exemple est donné dans la *leçon 2*, qui réutilise la classe du premier croquis comme bibliothèque. Un élément de bibliothèque se compose de deux fichiers ; le fichier d'en-tête (.h) contient la déclaration de la classe et le fichier source (.cpp) la définition des méthodes de la classe. Bien que cela ne soit pas strictement nécessaire, il vaut mieux utiliser le même nom pour les deux fichiers. Pour mettre notre classe `Virt_M` sous la forme d'une bibliothèque, la partie déclaration doit apparaître dans le fichier d'en-tête et la partie définition dans le fichier source. Pendant le développement, il est recommandé d'enregistrer les fichiers de la bibliothèque dans le même répertoire que le programme utilisateur. L'utilisation d'onglets dans l'EDI Arduino nous aide à cet égard, mais nous y reviendrons plus tard.

Au début d'un fichier d'en-tête, il est d'usage de vérifier s'il a déjà été inclus par le compilateur, ce qui est le cas si un autre fichier

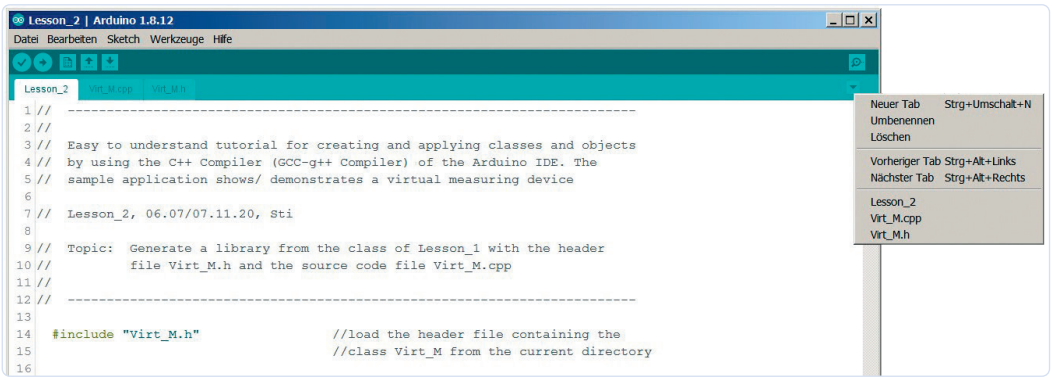


Figure 2. La fonction de gestion des onglets (leçon 2).

d'en-tête l'a déjà inclus. Des membres du même nom seraient alors déclarés plusieurs fois. La syntaxe pour que le préprocesseur le vérifie est la suivante :

```
  #ifndef Virt_M_h
// le symbole Virt_M_h est utilisé pour tester
//si le fichier d'en-tête a déjà été inclus
  #définir Virt_M_h // si le symbole n'existe pas
   encore, on le définit
    classe ....// déclaration de la classe
      { ....}; // fin de la déclaration
  #endif // fin de la compilation conditionnelle
```

L'instruction `#include "Virt_M.h"` dans le fichier source est utilisée pour inclure le fichier d'en-tête. Dans notre exemple, `Arduino.h` est inclus par défaut pour donner au compilateur l'accès aux bibliothèques standard. Avec cela, notre bibliothèque est terminée. Maintenant, pour utiliser la classe `Virt_M` dans un programme, il suffit de charger le fichier d'en-tête avec `#include "Virt_M.h"`.
Les bibliothèques Arduino fournissent souvent une instance de leur bibliothèque de classe. Par exemple, dans la bibliothèque Wire, *Wire* est un objet de la classe *TwoWire*. L'objet est inclus en utilisant `#include <Wire.h>`.

Une fois la programmation terminée, nous pouvons sauvegarder les fichiers de notre bibliothèque dans leur propre répertoire. Pour cela, Arduino crée le répertoire *libraries* dans le carnet de croquis. Nous pouvons y créer un autre répertoire dans lequel nous sauvegarderons nos fichiers. Cela fonctionne également via l'IDE avec *Croquis -> Inclure une bibliothèque -> Ajouter la bibliothèque .ZIP*.
Comme déjà mentionné, la fonction de gestion des onglets de l'IDE Arduino est très utile pour gérer plusieurs fichiers appartenant à un même projet. Elle est activée par le petit triangle en haut à droite de la fenêtre (**fig. 2**).

L'héritage - un concept fondamental de la POO

Une fois la POO abordée, le concept d'héritage n'est pas loin. Examinons-en les avantages. L'héritage consiste à mettre les attributs et les méthodes d'une classe de base à la disposition d'une classe héritière. La classe héritière peut être appelée classe enfant, classe dérivée ou sous-classe. La classe de base est également appelée classe parente ou ancêtre ou superclasse. La classe enfant ajoute d'autres propriétés à celles dont elle hérite et transmet le tout à sa propre descendance. Cela signifie que de nombreuses classes dérivées différentes peuvent être issues d'une classe parente, de sorte que la structure résultante n'est pas nécessairement linéaire, mais peut être arborescente. Une relation composée de nombreuses branches !
La *leçon 3* montre comment nous pouvons utiliser l'héritage dans notre tâche. Pour rendre l'exemple un peu plus significatif, nous utilisons des adresses multiplexées pour sélectionner les différents canaux de mesure et les lier aux objets créés. `Virt_M` est la classe de base. Ici, vous pouvez voir qu'il y a trois constructeurs avec des nombres différents de paramètres. Le C++ permet généralement cette *surcharge* des définitions de fonctions (c'est-à-dire des fonctions utilisant le même nom avec des nombres différents et/ou des types de paramètres différents). En POO, cela est connu sous le nom de *polymorphisme*. Notez le niveau d'accès `protected` qui indique que les méthodes, les classes et les autres membres sont accessibles à la classe héritière. Dans notre exemple, toutes les classes héritières ont accès à la méthode `set_Mux`.

Ici, la déclaration de la méthode contient aussi sa définition. Cela est possible, mais n'a d'intérêt que pour des fonctions très courtes. La première classe enfant est `Volt_M`, qui hérite de la classe `Virt_M`. La syntaxe est la suivante :

```
classe Volt_M : public Virt_M
// la classe Volt_M hérite de la classe Virt_M
```

Le mot-clé `public` garantit l'accès aux membres de `Virt_M`. `Volt_M` fournit à ses membres hérités l'attribut `sV`, le constructeur et la méthode `get_Unit`. La déclaration du constructeur n'apporte rien de nouveau, mais la définition, oui. Ici, le constructeur 2 de `Virt_M` est appelé. Lors de l'instanciation, les paramètres passés à `Volt_M` sont transmis à la classe `Virt_M`, la valeur initiale `ini_m` directement par le constructeur `Virt_M`, l'adresse multiplexée par la méthode `set_Mux` de `Volt_M`. La définition des constructeurs est la suivante :

```
Volt_M::Volt_M (byte v_mux, float ini_m):Virt_M(ini_m)
   // ini_m fourni par le constructeur de la classe
   Virt_M
   { set_Mux(v_mux); }     // v_mux par la méthode set_
```

La classe `Amp_M` est également une fille de `Virt_M`. Elle a une structure similaire à celle de `Volt_M`, sauf que le constructeur appelle le constructeur 3 de `Virt_M`. Le transfert complet des valeurs vers `Virt_M` est effectué sans aucune méthode supplémentaire. Il convient de noter ici qu'un héritage peut surcharger une méthode héritée.
Les classes sont maintenant entièrement déclarées, prêtes à l'utilisation. Les objets `M1`, `M2` et `M3` du type de classe `Virt_M` montrent l'appel au constructeur surchargé. `My_Volt_M` et `My_Amp_M` sont des objets des classes héritières `Volt_M` et `Amp_M`.

Après le démarrage du programme, le comportement de nos objets peut être suivi et compris grâce au moniteur série. Des accès invalides sont délibérément intégrés dans le croquis, mais sont initialement mis en commentaires dans l'exemple. En enlevant les marqueurs de commentaire `//` et en recompilant, nous pouvons étudier les messages d'erreur fournis par le compilateur.
Les objets enfants utilisent les structures héritées comme si elles faisaient partie d'eux-mêmes. Cependant, chaque objet possède sa propre copie des données. Tout se passe comme si le code de `Virt_M` avait été copié-collé dans `Volt_M` et `Amp_M`. Le principal inconvénient de ce type de relation (en plus d'un surcroît de travail et d'un code plus volumineux) est que toute modification de `Virt_M` entraîne des modifications chez toutes les classes qui la contiennent.

Une classe peut être aussi une amie

Dans le dernier exemple, nous avons vu qu'en raison de l'encapsulation, il n'est pas possible de modifier l'adresse multiplexée d'un objet. Cependant, surtout lorsque la programmation est proche du matériel, il est souvent souhaitable de disposer de nombreuses possibilités d'accès. Par exemple, si, lors du débogage nous voulons parcourir séquentiellement toutes les adresses multiplexées, nous devrons créer un objet pour chacune. Une autre possibilité serait de détruire l'objet après l'appel, puis de le recréer. Bien sûr, nous pourrions mettre la classe de base en accès public dès le début. Mais cela reviendrait à renoncer à l'un des avantages de la PPO, l'assurance d'un certain niveau de sécurité pour les données. La *leçon 4* nous montre une solution élégante à ce problème. Une nouvelle classe appelée `Debug_M` hérite de la classe

Listage 4. Déclaration de la classe Virt_M avec Debug_M comme classe « friend » (leçon 4).

```
class Virt_M
  {
    friend class Debug_M;               //la classe Debug_M obtient
                                        //l'accès aux membres privés
                                        //de Virt_M

    private:                            //membres privés de la classe
      float value=0;                    //valeur de mesure virtuelle, fixée à 0
      byte mux_adr= MUX_INVALID;        //adresse multiplexée pour la simulation

    public:                             //membres également accessibles depuis l'extérieur
      Virt_M (byte mux, float ini_m);   //constructeur avec valeurs initiales
      void set_Value (float val);       //méthode d'écriture des valeurs initiales
      float get_Value ();               //méthode de lecture de l'octet de mesure virtuel
      byte get_Mux ();                  //méthode de lecture de l'adresse mux
      String s="public: Demostring";    //pour démo seulement

  };                                    //Fin de la déclaration de la classe
```

Listage 5. Déclaration de la structure t_result (leçon 5).

```
struct t_result    //type pour retourner des données
  {
    byte mux;      //adresse mux
    String s;      //chaîne de caractères
    float val;     //valeur de mesure de type float
  };
```

Listage 6. Déclaration des attributs de la classe Dev_M (leçon 5).

```
class Dev_M
  {
    private:
      Volt_M *ptr_v;                    //pointeur vers un objet de type Volt_M
      Amp_M *ptr_a;                     //pointeur vers un objet de type Amp_M
      t_result result;                  //pour le retour de données
      const String sP=( "Power/ W   : ");
      const String sR= ("Resist/ Ohm: ");
```

de base `Virt_M` en tant qu'amie (`friend`) de cette classe. `Debug_M` est identique à `Volt_M`, sauf qu'elle a accès aux membres privés de `Virt_M`. Pour modifier l'adresse multiplexée, nous pouvons utiliser la méthode `set_` de `Debug_M` qui est déclarée `public`. Le **listage 4** montre la déclaration d'une classe `friend`.

La classe `Virt_M` ne contient pas de méthode pour définir l'adresse multiplexée. Au lieu de cela, le constructeur reçoit deux valeurs pour initialiser les attributs de la classe. L'une de ces valeurs est l'adresse multiplexée. Les paramètres sont transmis lorsque la classe de base et les sous-classes sont instanciées.

Une fois de plus, les informations de la routine `Setup` montrent les résultats de notre travail. Les classes amies sont rarement utilisées, bien qu'elles aient l'avantage d'utiliser des méthodes existantes, d'offrir les fonctions souhaitées sans compromettre le concept de sécurité de l'encapsulation lorsqu'elles sont utilisées correctement.

Accès d'une classe à une méthode d'une autre classe

Dans les derniers exemples, nous avons effectué des calculs de puissance et de résistance dans le programme principal. Nous avons lu les attributs de `Volt_M` et `Amp_M` via leurs méthodes. Nous allons maintenant le faire dans une classe qui leur est propre. La *leçon 5* définit la nouvelle classe `Dev_M`, qui accède aux méthodes des objets de `Volt_M` et `Amp_M`. Pour cela, lorsqu'une instance de `Dev_M` est créée, des pointeurs vers les objets à appeler sont passés par le constructeur à `Dev_M`.

Mais d'abord, en utilisant `struct`, nous créons un type de données nommé `t_result` (**listage 5**), avec plusieurs membres pour contenir les résultats. Ensuite, nous déclarons la classe `Dev_M` (**listage 6**). Les pointeurs `ptr_v` et `ptr_a`, qui sont utilisés dans les classes `Volt_M` et `Amp_M`, sont initialisés par le constructeur (**listage 7**). L'opérateur

Listage 7. Le constructeur de Dev_M enregistre les pointeurs (leçon 5).

```
    // Le constructeur passe les pointeurs aux objets de Volt_M, Amp_M
    //----------------------------------------------------------------

          Dev_M::Dev_M (Volt_M *ptrv, Amp_M *ptra)
            {
               ptr_v= ptrv;              //pointeur vers un objet de Volt_M
               ptr_a= ptra;              //pointeur vers un objet de Amp_M
            }
```

Listage 8. Définition de la méthode get_Powr de Dev_M (leçon 5).

```
t_result Dev_M::get_Powr ()               //calcul de la puissance
  {
    result.s= sP;                         //nom, unité
    result.val= ptr_v -> get_Value () * ptr_a -> get_Value ();
    return result;                        //données retournées dans une structure t_result
  }
```

Listage 9. Instanciation des objets, passage des pointeurs aux objets (leçon 5).

```
  Volt_M My_Volt_1 (V_MUX0, 1.00);          //objet de type Volt_M
  Volt_M My_Volt_2 (V_MUX1, 2.00);          //objet de type Volt_M
  Amp_M My_Amp_1 (A_MUX0, 1.10);            //objet de type Amp_M
  Amp_M My_Amp_2 (A_MUX1, 2.10);            //objet de type Amp_M
  Dev_M My_Dev_1 (&My_Volt_1, &My_Amp_1);   //objet de type Dev_M, passage de pointeurs
  Dev_M My_Dev_2 (&My_Volt_2, &My_Amp_2);   //objet de type Dev_M, passage de pointeurs
```

'`*`' indique que le nom suivant est interprété comme un pointeur. Les méthodes `Dev_M` peuvent maintenant utiliser les pointeurs pour accéder aux objets externes. L'accès est réalisé en utilisant une instruction de la forme `pointeur->méthode`. L'opérateur '`->`' à la place de l'opérateur '`.`' (point) permet d'appeler une méthode en utilisant un pointeur (**listage 8**).
Pour pouvoir créer une instance de `Dev_M`, il est nécessaire de créer au préalable de nouvelles instances des classes référencées. Celles-ci doivent être du type `Volt_M` et `Amp_M`. Lors de l'instanciation de `Dev_M`, nous déclarons les pointeurs vers ces objets dans la liste des paramètres du constructeur en utilisant l'opérateur '`&`' (**listage 9**). Comme le montre la sortie de la routine `Setup` (**fig. 3**) de notre application, les appels des méthodes des objets `My_Dev_1` et `My_Dev_2` fournissent tous les résultats.

Héritage multiple

Dans l'exemple précédent, nous avons d'abord dû créer deux instances distinctes avant de pouvoir utiliser les méthodes de `Dev_M`. Il y a sans doute des applications où cela a un sens, mais ici, c'était surtout pour illustrer l'usage de pointeurs vers des objets. Les classes de base et les classes enfants peuvent être héritées directement par une autre classe comme le montre la *leçon 6* ; c'est l'*héritage multiple*. Les deux classes dérivées `Volt_M` et `Amp_M` sont héritées par la classe de base `Dev_M`. Comme les classes dérivées contiennent `Virt_M`, `Dev_M` a accès aux membres de toutes les classes.

Afin de renvoyer simplement les résultats complexes des méthodes `Dev_M`, nous déclarons à nouveau le type de données `t_result`. Lorsque la classe est créée, les classes héritées sont listées après l'opérateur '`:`', séparées par des virgules :

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

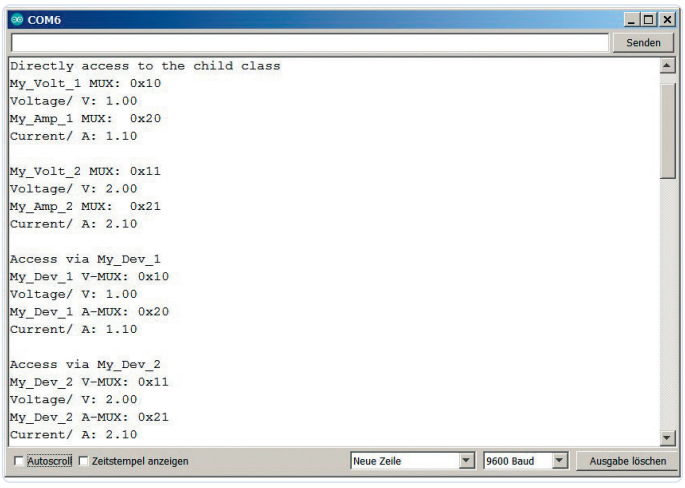


Figure 3. Sortie de la routine Setup de la leçon 5.

Listage 10. Référence explicite à un membre en cas d'ambiguïté (leçon 6).

```
t_result Dev_M::get_Volt ()             //lecture de la tension courante
  {
    result.s= Volt_M::get_Unit ();      //appel de la méthode de Volt_M
    result.val= Volt_M::get_Value ();   //appel de la méthode de Volt_M
    return result;
  }
```

Listage 11. Ambiguïté de la méthode get_Mux (leçon 6).

```
  Serial.print ("My_Dev_1, Volt MUX: 0x"),
  Serial. println (My_Dev_1.Volt_M::get_Mux(), HEX);  //méthode de la classe de base
  Serial.print (My_Dev_1.get_Volt().s);
  Serial.println (My_Dev_1.get_Volt().val);           //valeur initiale de l'instanciation de My_Dev_1
  Serial.println ();
```

Listage 12. Utiliser ses propres méthodes pour lever l'ambiguïté (leçon 6).

```
  Serial.print ("My_Dev_2, Volt MUX: 0x");
  Serial.println (My_Dev_2.get_VMux(),HEX);   //méthode de la classe héritée
  Serial.print (My_Dev_2.get_Volt().s);
  Serial.println (My_Dev_2.get_Volt().val);   //valeur initiale de l'instanciation de My_Dev_M
  Serial.println ();
```

Les paramètres à transmettre aux classes héritées sont énoncés dans une liste de paramètres en fin de déclaration du constructeur :

```
Dev_M (byte v_x, float i_v, byte a_x, float i_a);
  // le constructeur de la classe Dev_M
```

La définition du constructeur attribue ensuite les valeurs de transfert aux paramètres. A l'instanciation de `Dev_M`, les constructeurs de `Volt_M` et `Amp_M` sont appelés et les attributs de ces classes sont initialisés avec les paramètres.

```
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) {}
```

Comme déjà mentionné, nous pouvons accéder à tous les attributs et méthodes des classes héritières via `Dev_M`, à condition que la spécification d'accès le permette. `Dev_M` nous fournit les méthodes appropriées pour ce faire. Il y a cependant un problème : comme les deux classes héritées sont elles-mêmes dérivées de la classe `Virt_M`, elles ont des méthodes et des attributs homonymes, ce qui conduit à des ambiguïtés qu'on lève en utilisant l'opérateur de résolution de portée '`:`' pour spécifier explicitement de quel membre de quelle classe il s'agit (**listage 10**).
Nos classes sont maintenant définies et nous pouvons en instancier les objets. En principe, les méthodes de `Dev_M` sont tout à fait suffisantes pour écrire une application. Pour une meilleure compréhension, nous créons également des instances supplémentaires de la classe de base et des classes dérivées. Comme une adresse multiplexée différente est attribuée à chaque objet lors de l'instanciation, cela nous indique quel objet est actuellement adressé.
Après le démarrage de l'application, le moniteur série affiche les résultats des appels aux méthodes. L'exécution de la routine `Setup` est intéressante. Les attributs de la classe de base sont lus de différentes manières. L'accès à l'adresse multiplexée de l'objet `My_Dev_1` appelle la méthode de la classe de base. On a le même problème d'ambiguïté qu'avec les définitions des méthodes de `Dev_M`. Pour lire l'adresse multiplexée, on doit préciser explicitement si c'est celle de la tension ou du courant. La méthode est appelée `get_Mux` dans les deux cas (**listage 11**). Le problème est résolu en spécifiant la référence de la classe.
Il est bien sûr beaucoup plus clair et finalement plus simple de fournir leurs propres méthodes aux classes enfants `Volt_M` et `Amp_M` (**listage 12**). Dans l'exemple, ces méthodes sont respectivement `get_VMux` et `get_AMux`. Elles sont utilisées lors de la lecture des attributs de `My_Dev_2`.
La fonction `sizeof` sert à déterminer le nombre d'octets dans les champs de données de notre objet. La classe de base alloue 11 octets tandis que les classes dérivées en utilisent 17. Comme la classe de base est incluse dans chaque classe dérivée, les classes dérivées ajoutent 6 octets chacune. `Dev_M` hérite deux fois de 17 octets et nécessite elle-même 22 octets ; on arrive à un total de 56 octets (**fig. 4**).

Un régal pour les fans d'Arduino

Je suis sûr qu'il y a des fans d'Arduino qui ont souvent souhaité pouvoir intégrer une des méthodes Arduino dans leur programme. Peut-être avez-vous développé votre propre afficheur que vous

aimeriez gérer avec la puissante méthode `print`. Comme déjà suggéré, cela peut être réalisé en créant une classe qui hérite de la classe `Print`. Le croquis de la *leçon 7* montre comment faire. La classe `My_Print_C` hérite de toutes les méthodes de `Print` et surcharge la méthode `write` originale, qui normalement envoie les données d'affichage sur un matériel spécifique. La méthode `write` reçoit un seul caractère de la méthode `print(ln)` comme argument, qu'elle transfère ensuite à l'afficheur via une interface matérielle, telle qu'un bus I²C. Notre exemple n'utilise aucun matériel : les caractères transférés sont écrits dans une chaîne dont le contenu peut être visualisé sur le moniteur série.
Comme les méthodes `print(ln)` sont surchargées, le type de valeurs que nous pouvons « imprimer » n'a pratiquement aucune importance. C'est tout. Nous pouvons maintenant utiliser `print` pour envoyer des caractères à notre propre afficheur.

Conclusion

J'espère que cet article vous a fourni une bonne base pour débuter dans la programmation orientée objet. Nous avons examiné les bibliothèques de classes et la manière de les intégrer dans nos applications. Dans le dernier exemple, nous avons montré qu'hériter de classes externes et les modifier n'est pas un problème. Même s'il reste encore beaucoup à apprendre, les bases pour écrire vos propres programmes orientés objet devraient maintenant toutes être en place.
J'espère vous avoir convaincus des avantages de la POO sur la programmation procédurale. Ces notions peuvent être difficiles à maîtriser au début, surtout si vous êtes un familier de la programmation procédurale. Mais courage, c'est en forgeant qu'on devient forgeron !

(200563-04)

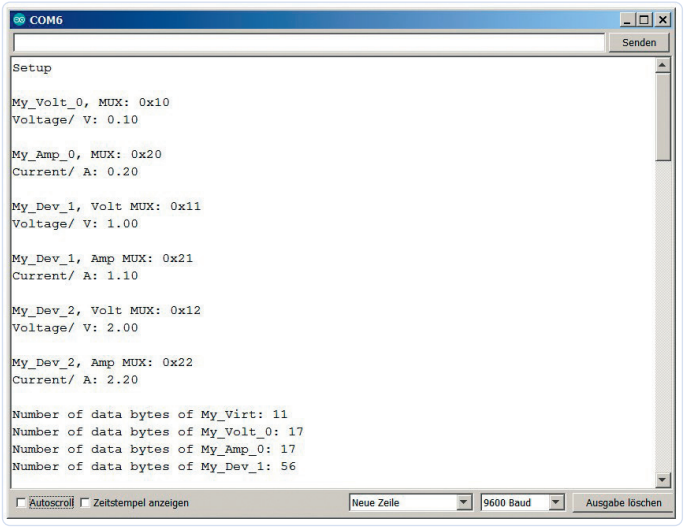
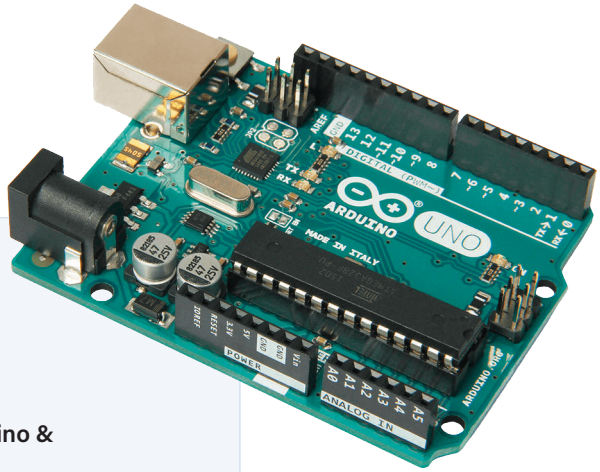


Figure 4. Sortie de la routine Setup de la leçon 6.



Contributeurs
Idée et texte : **Roland Stiglmayr**
Rédaction : **Rolf Gerstendorf**
Mise en page : **Giel Dols**
Traduction : **Helmut Müller**

Des questions, des commentaires ?
Envoyez un courriel à l'auteur (1134-715@online.de) ou contactez Elektor (redaction@elektor.fr).

LIENS

[1] **Page de cet article** : www.elektormagazine.fr/200563-04
[2] **Programmation en C++ (en anglais)** : https://en.wikibooks.org/wiki/Subject:C%2B%2B_programming_language
[3] **Séminaire à l'Université technique de Munich** : www.ei.tum.de/fileadmin/tueifei/ldv/Vorlesungen/cpp/CPP-Skript.pdf