Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

a.09 – Programmation OO en C

Classes T-2/I-2 // 2018-2019

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Daniel Gachet | HEIA-FR/TIC
a.09 | 10.09.2018

Contenu

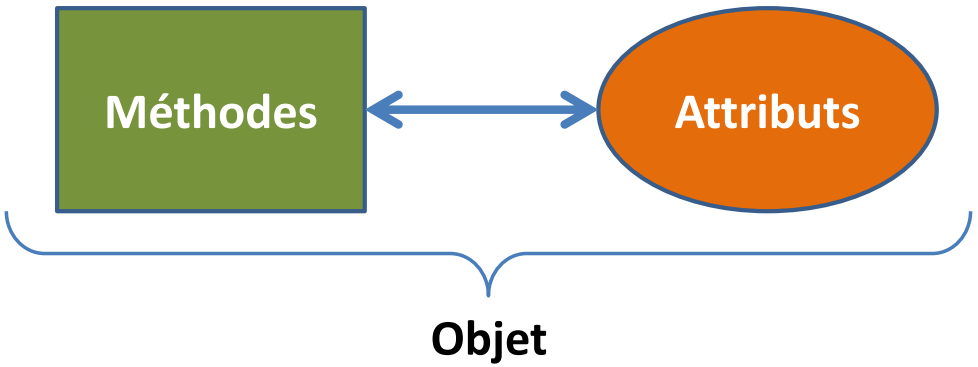
- Introduction
- Concepts
- Héritages
- Relations

- **Le langage de programmation C ne supporte pas le concept de programmation orientée objet. Cependant, il est tout à fait possible de programmer en C dans un style OO. Pour cela il suffit de revenir au concept de base de la programmation OO.**

- **Qu'est ce qu'un objet en langage OO:**

  - des attributs (un structure de données) définissant l'état de l'objet

  - des méthodes (des algorithmes) définissant le comportement de l'objet



- **Pour déclarer/décrire un objet en C il suffit de:**
  - regrouper tous les attributs décrivant l'état d'un objet dans une structure
  - regrouper les méthodes permettant d'agir sur l'objet dans la même structure que les données (pointeurs de fonctions)
  - de passer comme argument la référence de la structure à toutes les méthodes (le pointeur sur la structure / sur l'objet)

```
struct my_first_c_class {
   void (*m_method_1) (struct my_first_c_class*, int);
   int  (*m_method_2) (struct my_first_c_class*);
   int m_attribute_1;
   int m_attribute_2;
};
```

  - Des méthodes globales peuvent également être utilisés comme façade pour agir sur des objets. Pour ces méthodes, la référence de la structure doit également être passée comme argument.

- Les méthodes agissant sur les données d'un objet s'implémentent de la même manière que des routines C classiques, p. ex.:

```
void method_1(struct my_first_c_class* oref, int arg)
{
   oref->m_attribute_1 = arg + oref->m_attribute_2;
}


int method_2(struct my_first_c_class* oref)
{
   return oref->m_attribute_1;
}
```

- Ces méthodes peuvent également être déclarées dans le header file afin de pouvoir être utilisées par les classes dérivées.

Instanciation

- **Il est bien évident que le langage C ne connaît pas le concept de constructeur et de destructeur. Pour palier à ce manque, on implémentera simplement une ou deux fonctions les simulant.**

- **Implémenation du constructeur, p. ex.:**
  - Header-file

```
extern void init_my_first_c_class (my_first_c_class* oref);
```

  - Implementation-file

```
void init_my_first_c_class (my_first_c_class* oref)
{
  oref->m_method_1 = method_1;
  oref->m_method_2 = method_2;
  oref->m_attribute_1 = 0;
  oref->m_attribute_2 = 20;
}
```

- **En dynamique**
  - Déclaration de l'objet

```
struct my_first_c_class* object;
```

  - Instanciation et initialisation de l'objet

```
object = malloc (sizeof(*object));
init_my_first_c_class (object);
```

  - Utilisation

```
object->m_method_1(object, 100);
int val = object->m_method_2(object);
```

- **En statique**
  - Déclaration de l'objet et instanciation

```
struct my_first_c_class object;
```

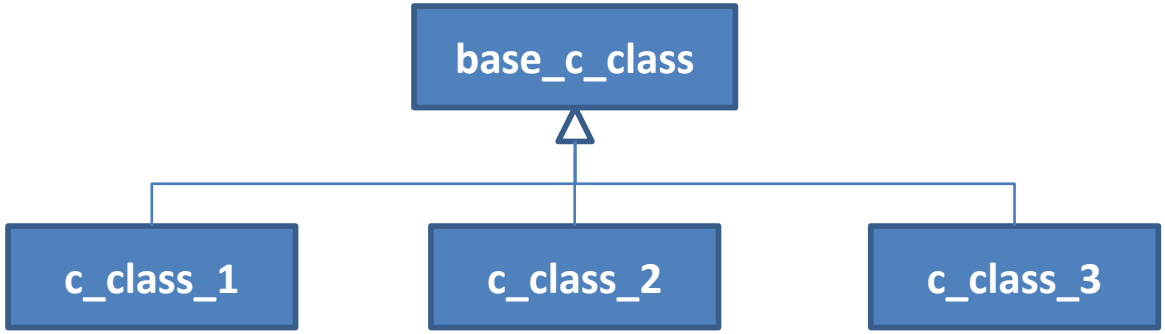
  - Initialisation de l'objet

```
init_my_first_c_class (&object);
```

  - Utilisation

```
object.m_method_1(&object, 100);
int val = object.m_method_2(&object);
```

- **Dans le concept orienté objet il est bien évident que la notion d'héritage est un élément essentiel.**



- **En C celui-ci peut être réalisé par ajout (composition) de la classe de base dans les classes dérivée ou, plus exactement, de la structure de base dans les structures dérivées.**

Héritage – déclaration

- **Déclaration de la classe de base**

```
struct base_c_class {
    int (*m_method_1) (struct base_c_class*, int);
    int (*m_method_2) (struct base_c_class*, int, int);
    int m_attribute_1;
    int m_attribute_2;
};
```

- **Déclaration de la classe dérivée**

```
struct c_class_1 {
    int (*m_method_3) (struct c_class_1*, int);
    struct base_c_class m_base; /* → dérivation */
    int m_attribute_3;
};
```

  - Attention: m_base ne doit pas être le pointeur sur sa structure.

Héritage – initialisation

- **Initialisation de la classe de base**

```
void init_base_c_class (struct base_c_class* oref)        {
  oref->m_method_1 = bc_method_1;
  oref->m_method_1 = bc_method_2;
  oref->m_attribute_1 = 0;
  oref->m_attribute_2 = 0;
}
```

- **Initialisation de la classe dérivée**

```
void init_c_class_1 (struct c_class_1* oref)
 {
   oref->m_method_3 = c1_method_3;
   init_base_c_class (&oref->m_base);
   oref->m_base.m_method_1 = c1_method_1; /* → overloading */
   oref->m_attribut_3 = 0;
 }
```

Héritage – implémentation d'une méthode polyphorme

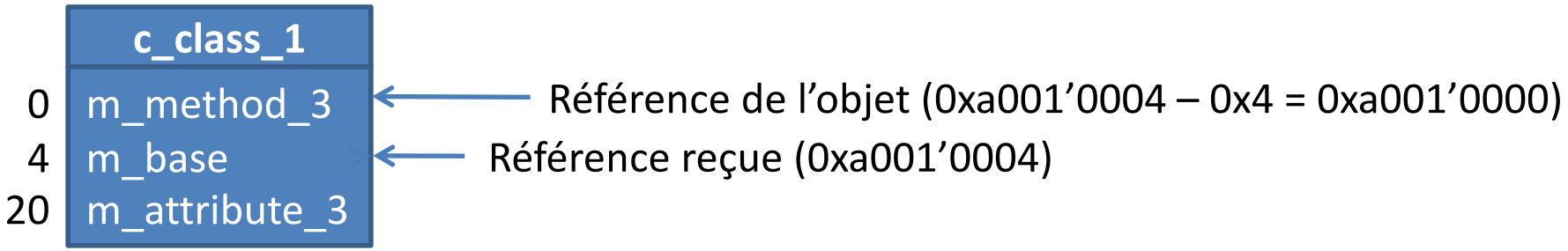
- Implémentation de la méthode 1 de la classe de base

```
int bc_method_1 (struct base_c_class* oref, int arg)
{
  return oref->m_attribute_1 + arg;
}
```

- Implémentation de la méthode 1 de la classe dérivée

```
int c1_method_1 (struct base_c_class* oref, int arg)
{
   struct c_class_1* c1_ref =
     container_of(oref, struct c_class_1, m_base);
   return c1_ref->m_attribute_3 + arg;
}
```

- **La référence sur l'objet d'une classe dérivée peut être calculée à partir d'un des attributs de la classe passé comme référence. Pour cela il suffit de soustraire l'offset de cet attribut dans la structure à la référence reçu.**



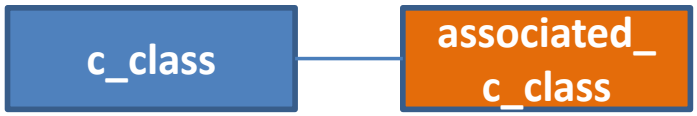
- **Ce calcul peut être réalisé avec la macro suivante:**

```
#define container_of(ptr, type, member)\
((type*) ((char*)(ptr) - offset_of(type, member)))

#define offset_of(type, member)\
((char*)&(((type*)0)->member))
```

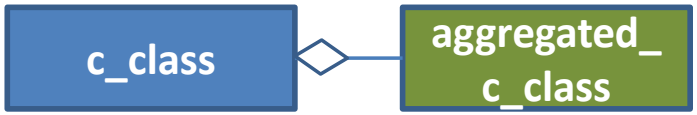
- **Les 3 différents types de relations (association, agrégation et composition) peuvent simplement être réalisés en C selon les modèles ci-dessous:**

  - Association



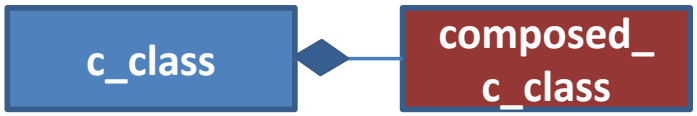

```
struct c_class {
  struct associated_c_class* m_association;
  /* ... */
};
```

  - Agrégation



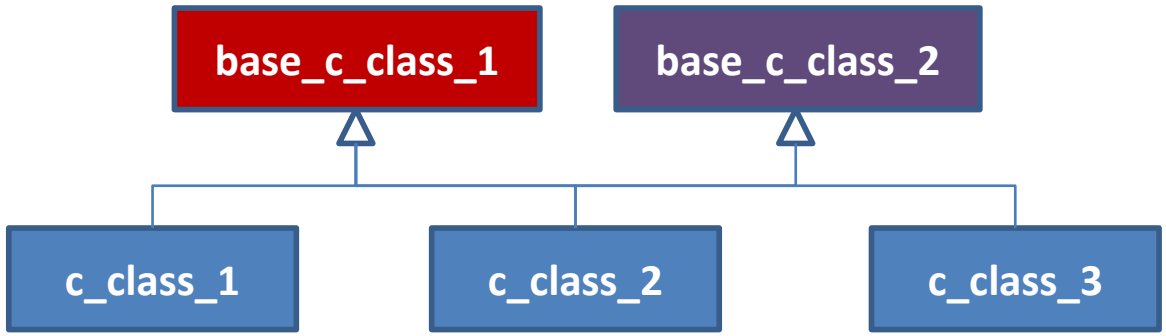

```
struct c_class {
  struct aggregated_c_class* m_aggregation;
  /* ... */
};
```

  - Composition




```
struct c_class {
  struct composed_c_class m_composition;
  /* ... */
};
```

- La technique d'héritage simple peut naturellement être généralisée pour permettre des héritages multiples.




```
struct c_class_1 {
    /* additional methods…. */

    struct base_c_class_1 m_base_1;
    struct base_c_class_2 m_base_2;

    /* additional member attributes */
};
```