

La programmation orientée objet en C avec GObject

Première partie : les classes
par Nicolas JOSEPH ([home](#)) ([Blog](#))

Date de publication : 10 Janvier 2008

Dans ce premier tutorial consacré à la bibliothèque GObject, nous allons aborder les notions essentielles qui vous permettront de créer vos premiers objets en C.

Public concerné

I - Introduction

II - Le code minimal

III - Champs public

 III-A - Les méthodes

 III-B - Les variables

IV - Champs privés

 IV-A - Les méthodes

 IV-B - Les variables

V - Constructeur

VI - Destructeur

VII - Membres de classes

VIII - Les méthodes virtuelles

IX - Les propriétés

X - GObject Builder

XI - Exemple complet



XII - Conclusion

XIII - Remerciements

Public concerné



I - Introduction

A l'origine, le langage C est dépourvu de fonctionnalité permettant de faire de la  **programmation orientée objet** (POO), il est au mieux possible de créer des  **types abstrait de données** (TAD). Cependant lorsque l'on connaît parfaitement le langage C et moyennant quelques lignes de code supplémentaires, il est possible d'obtenir les mêmes fonctionnalités qu'en C++, par exemple.

II - Le code minimal

Le C n'offrant pas les mêmes facilités que le C++, ou autres langages orientés objet, même avec l'aide de la bibliothèque GObject, il va falloir écrire plus de code.

Avant toute chose, il faut préciser qu'une classe, avec GObject, est en fait divisée en deux parties. La première portant le nom de la classe à proprement dit (dans notre exemple *Vehicule*) et est relative aux instances de notre classe (autrement dit les objets). Elle contiendra donc les variables publiques et privées.

La seconde partie porte le nom de la classe avec le suffixe *Class* (dans notre exemple *VehiculeClass*) qui contiendra l'ensemble des informations relatives à notre classe (typiquement les méthodes et variables de classes).

Même si le rôle de ces deux parties n'est pas clair, gardez en tête cette séparation, la compréhension de la suite n'en sera que plus rapide.

Voici donc le code minimal afin de créer une nouvelle classe :

vehicule.h

```
#ifndef H_NJ_VEHICULE_261120071937
#define H_NJ_VEHICULE_261120071937

#include <glib-object.h>

#define TYPE_VEHICULE (vehicule_get_type ())
#define VEHICULE(obj) (G_TYPE_CHECK_INSTANCE_CAST ((obj), TYPE_VEHICULE, Vehicule))
#define VEHICULE_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST ((klass), TYPE_VEHICULE, VehiculeClass))
#define IS_VEHICULE(obj) (G_TYPE_CHECK_INSTANCE_TYPE ((obj), TYPE_VEHICULE))
#define IS_VEHICULE_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass), TYPE_VEHICULE))
#define VEHICULE_GET_CLASS(obj) (G_TYPE_INSTANCE_GET_CLASS ((obj), TYPE_VEHICULE, VehiculeClass))

typedef struct _Vehicule Vehicule;
typedef struct _VehiculeClass VehiculeClass;

struct _Vehicule
{
    GObject parent;
};

struct _VehiculeClass
{
    GObjectClass parent;
};

GType vehicule_get_type (void);

#endif /* H_NJ_VEHICULE_261120071937 */
```

vehicule.c

```
#include "vehicule.h"

GType vehicule_get_type (void)
{
    static GType type = 0;

    if (type == 0)
    {
        static const GTypeInfo info = {
            sizeof (VehiculeClass), /* class_size */
            NULL, /* base_init */
            NULL, /* base_finalize */
```

vehicule.c

```
NULL,          /* class_init */
NULL,          /* class_finalize */
NULL,          /* class_data */
sizeof (Vehicule), /* instance_size */
0,            /* n_preallocs */
NULL,          /* instance_init */
NULL          /* value_table */
};
type = g_type_register_static (G_TYPE_OBJECT, "Vehicule", &info, 0);
}
return type;
}
```

Dans le fichier d'en-tête nous déclarons un ensemble de macro qui doivent vous être familière si vous avez déjà utilisé GTK+ :

- **TYPE_VEHICULE** : il s'agit de notre nouveau type. Cette macro sera utilisée afin d'en créer une nouvelle instance,
- **VEHICULE** : cette macro est utilisée afin de faire un cast d'un objet vers notre type,
- **VEHICULE_CLASS** : c'est la même chose mais pour la classe,
- **IS_VEHICULE** : vérifie que l'objet est du type souhaité,
- **IS_VEHICULE_CLASS** : idem pour la classe,
- **VEHICULE_GET_CLASS** : permet de récupérer la structure représentant la classe à partir d'un objet.

Ensuite nous déclarons une première structure qui représente un objet de notre classe. C'est un pointeur sur cette structure que nous obtiendrons lorsque l'on créera un nouvel objet. Le premier membre de cette structure doit **impérativement** être la classe mère.

La seconde partie, qui est commune à toutes les instances de notre classe, est déclarée de manière similaire, en commençant par la classe parent.

Pour finir, nous déclarons la seule fonction, qui est en faite une fonction privée qui ne doit pas être utilisée directement (il faut passer par la macro **TYPE_VEHICULE**) qui permet de connaître le type que GObject à associé à notre classe. Lors du premier appel à cette fonction, nous allons enregistrer notre nouveau type. Pour ce faire, il faut commencer par renseigner une structure de type **GTypeInfo** :

```
typedef struct {
    /* interface types, classed types, instantiated types */
    guint16          class_size;

    GBaseInitFunc    base_init;
    GBaseFinalizeFunc base_finalize;

    /* interface types, classed types, instantiated types */
    GClassInitFunc   class_init;
    GClassFinalizeFunc class_finalize;
    gconstpointer    class_data;

    /* instantiated types */
    guint16          instance_size;
    guint16          n_preallocs;
    GInstanceInitFunc instance_init;

    /* value handling */
    const GTypeValueTable *value_table;
} GTypeInfo;
```

Pour commencer, nous avons juste besoin de préciser les tailles des structures de classe et d'instance.

Nous verrons en fonction de nos besoins l'utilité des autres éléments.

Une fois notre classe créée, il ne reste plus qu'à la tester :

```
#include "vehicule.h"

int main (void)
{
    Vehicule *v = NULL;

    g_type_init ();
    v = g_object_new (TYPE_VEHICULE, NULL);
    g_object_unref (v);
    return 0;
}
```

Nous ne pouvons pas faire grand chose avec notre classe, mais vous pouvez déjà remarquer que nous pouvons utiliser les fonctions spécifiques aux classes de base (GObject) sans problème.

Maintenant que nous avons vu en détails la création d'une classe, il existe une macro afin de nous éviter le fastidieux travail de copier/coller :

```
G_DEFINE_TYPE (Vehicule, vehicule, G_TYPE_OBJECT)
```

 *Il ne faut pas de point-virgule puisque la macro est dépliée sous la forme :*

```
static void vehicule_init (Vehicule *self);
static void vehicule_class_init (VehiculeClass *klass);
static gpointer vehicule_parent_class = NULL;

static void vehicule_class_intern_init (gpointer klass)
{
    vehicule_parent_class = g_type_class_peek_parent (klass);
    vehicule_class_init ((VehiculeClass*) klass);
}


GType vehicule_get_type (void)
{
    static GType g_define_type_id = 0;

    if (G_UNLIKELY (g_define_type_id == 0))
    {
        g_define_type_id =
            g_type_register_static_simple (TYPE_PARENT,
                                           g_intern_static_string ("Vehicule"),
                                           sizeof (VehiculeClass),
                                           (GClassInitFunc)vehicule_class_intern_init,
                                           sizeof (Vehicule),
                                           (GInstanceInitFunc)vehicule_init,
                                           (GTypeFlags) flags);
    }
    return g_define_type_id;
}
```

Il nous reste donc plus qu'à définir les deux fonctions initialisations (éventuellement vide), avec un prototype légèrement différent, puisque grâce à la macro, GObject connaît le vrai type de notre classe :

```
static void vehicule_class_init (VehiculeClass *klass)
{
    g_return_if_fail (klass != NULL);
}

static void vehicule_init (Vehicule *self)
{
    g_return_if_fail (self != NULL);
}
```

 Notez au passage que nous obtenons aussi la classe parent de notre classe, ceci nous sera utile par la suite pour le chaînage des fonctions.

III - Champs public

III-A - Les méthodes


Les méthodes publiques sont de simples fonctions externes. Par convention, ces fonctions ont un nom qui commence par celui de la classe et leur premier argument est l'objet auquel doit s'appliquer la fonction. C'est le *this* ou *self* passé implicitement dans langages orientés objets.

III-B - Les variables

Pour ajouter une variable publique à notre classe, il suffit de l'ajouter à la structure qui la définit :

```
struct _Vehicule
{
    GObject parent;

    /*< public >*/
    guint32 flags;
};
```

 *La structure parent doit toujours être le premier membre de la structure. Nous ajoutons donc nos variables en dessous.*

Pour accéder à notre champ, il suffit de procéder comme pour une structure classique :

```
v->flags = 0;
g_print ("%d\n", v->flags);
```

IV - Champs privés

IV-A - Les méthodes

Nous pouvons aussi définir des méthodes privées à notre classe, simplement en les définissant dans le fichier source en tant que fonctions statiques :

```
static void vehicule_reset_flags (Vehicule *self)
{
    g_return_if_fail (self != NULL);
    g_return_if_fail (IS_VEHCULE (self));

    self->flags = 0;
}
```

IV-B - Les variables

Pour définir des variables privées, il ne suffit pas de déclarer une variables *static*, puisque dans ce cas elle serait partagée par l'ensemble des instances d'une classe. GObject propose un mécanisme qui permet à chaque objet de disposer de ses propres variables privées.

Pour cela, il faut créer une structure qui regroupera l'ensemble des variables privées :

```
typedef struct _VehiculePrivate VehiculePrivate;

struct _VehiculePrivate
{
    gint vitesse;
};
```

Ensuite, nous demandons à GObject, d'enregistrer cette structure lors de l'initialisation de notre classe :

```
static void vehicule_class_init (gpointer g_class, gpointer class_data)
{
    /* ... */
    g_type_class_add_private (klass, sizeof (VehiculePrivate));
    /* ... */
}
```

Et enfin pour y accéder à partir d'une instance de notre classe, nous créons une macro :

```
#define VEHCULE_GET_PRIVATE(o) (G_TYPE_INSTANCE_GET_PRIVATE ((o), \
    TYPE_VEHCULE, \
    VehiculePrivate))
```

Et pour y accéder :

```
VehiculePrivate *priv = NULL;

priv = VEHCULE_GET_PRIVATE (self);
g_print ("%d\n", priv->vitesse);
```

V - Constructeur

Le constructeur et le destructeur sont automatiquement appelés par GObject lorsque cela est nécessaire. Comme nous ne devons pas les appeler directement, il faut les enregistrer lors de l'initialisation de notre classe :

```
static void vehicule_class_init (gpointer g_class, gpointer class_data)
{
    GObjectClass *gobject_class = G_OBJECT_CLASS (g_class);

    gobject_class->constructor = vehicule_constructor;

    parent_class = g_type_class_peek_parent (g_class);
    (void)class_data;
}
```

Et dans le constructeur, nous devons commencer par appeler le constructeur de la classe parent :

```
static GObject *vehicule_constructor (GType type, guint n_construct_properties,
                                     GObjectConstructParam *construct_properties)
{
    GObject *obj = NULL;


    /* Appel du constructeur de la classe parent */
    obj = parent_class->constructor (type,
                                     n_construct_properties,
                                     construct_properties);

    /* ... */
    return obj;
}
```

VI - Destructeur

Pour la destruction d'un objet, le principe est le même, mis à part que l'on dispose de deux fonctions qui peuvent être surchargées :

- `dispose` : sert à libérer la mémoire allouée au cours de la vie de notre objet. Par contre l'objet lui même n'est pas détruit,
- `finalize` : il s'agit du véritable destructeur de notre objet, qui commence par un appel à la méthode `dispose`.

 *Contrairement au constructeur, le chaînage (l'appel au destructeur de la classe parent), doit se faire en dernier.*

VII - Membres de classes

Les membres de classes (méthodes et variables) sont partagés par l'ensemble des objets de la classes. En C++, ils sont déclarés à l'aide du mot-clé *static*. Pour obtenir le même résultat, nous avons juste à ajouter un membre dans la structure qui définit notre classe :

```
struct _VehiculeClass
{
    GObjectClass parent;

    /*< private >*/
    gint static_integer;
}
```

Vous remarquerez que nous faisons appel au bon sens des développeurs pour ce qui est de la visibilité des variables de classes.

Pour accéder à notre variable, nous devons récupérer notre classe. Si nous en possédons une instance, rien de plus simple, il suffit d'utiliser la macro que nous avons écrite au dessus :

```
VehiculeClass *klass = VEHICULE_GET_CLASS (self);
```

Mais il est aussi possible d'obtenir notre classe en l'absence d'instance de celle-ci (c'est la définition même des membres statiques). Pour ce faire, il suffit d'utiliser la fonction *g_type_class_peek* :

```
VehiculeClass *klass = g_type_class_peek (TYPE_VEHICULE);
```

VIII - Les méthodes virtuelles

Les méthodes virtuelles est un concept essentiel à la POO et si vous avez été attentif nous avons déjà utilisé ce concept dans nos exemples précédents. Où ? Tout simplement pour la construction et la destruction de nos objets.

Pour créer une méthode virtuelle, il faut commencer par déclarer un pointeur de fonction dans notre classe :

```
struct _VehiculeClass
{
    GObjectClass parent;

    void (* avancer) (Vehicule *self, gint distance);
};
```

Ce pointeur nous permettra d'appeler la bonne fonction pour faire avancer notre véhicule quelque soit la classe de base de l'objet.

Si vous souhaitez créer une méthode publique, il suffit de créer un assesseur public :

```
void vehicule_avancer (Vehicule *self, gint distance)
{
    VEHICULE_CLASS (self)->avancer (self, distance);
}
```

Sinon, comme pour les méthodes, utilisez le mot clés *static* pour rendre cet assesseur, et donc votre méthode virtuelle, privé.

Il ne faut pas oublier d'initialiser notre pointeur dans la fonction d'initialisation de notre classe :

```
static void vehicule_rouler (Vehicule *self, gint distance)
{
    /* ... */
}

static void vehicule_class_init (VehiculeClass *klass)
{
    g_return_if_fail (klass != NULL);

    klass->avancer = vehicule_rouler;
}
```

L'avantage c'est que pour notre nouvelle classe *robot* qui est un véhicule qui n'avance pas en roulant mais en marchant, nous avons juste à écrire :

```
static void robot_marcher (Vehicule *self, gint distance)
{
    /* ... */
}

static void robot_class_init (RobotClass *klass)
{
    g_return_if_fail (klass != NULL);

    VEHICULE_CLASS (klass)->avancer = robot_marcher;
}
```

Et dans les deux cas, l'appel à la fonction *vehicule_avancer* fera avancer notre véhicule avec la bonne méthode !

 *Il est possible d'initialiser une méthode virtuelle à NULL, dans ce cas c'est aux classes filles de les définir. On parle alors de [FAQ](#) **méthodes virtuelles pures**.*

IX - Les propriétés

En POO, les propriétés sont semblables aux variables dans leur utilisation. Cependant dans le cas des propriétés le fait d'accéder ou de modifier sa valeur fait appel à une fonction ce qui permet d'avoir une abstraction supplémentaire. Par exemple nous pourrions avoir une propriété dont la valeur est le résultat d'une requête (et donc sa modification entraînera une mise à jour de la base de données).

En plus d'un niveau d'abstraction supplémentaire, il est possible d'avoir des propriétés accessible uniquement en lecture ou en écriture.

Pour commencer chaque propriété doit avoir un identifiant unique au sein de la classe, le plus simple étant de créer une énumération :

```
enum
{
    PROP_0,
    PROP_WINDOW_OPEN
};
```

La première valeur d'une énumération étant zéro, nous utilisons un nom bidon pour que notre première propriété ait l'identifiant 1. Comme son nom l'indique notre unique propriété permet de savoir si la fenêtre de notre véhicule est ouverte. Logiquement, modifier cette propriété permettra d'ouvrir ou non la fenêtre.

Nous avons donc une propriété de type *boolean* que nous définissons dans la fonction d'initialisation de notre classe :

```
static void vehicule_class_init (gpointer g_class, gpointer class_data)
{
    GObjectClass *gobject_class = G_OBJECT_CLASS (g_class);
    GParamSpec *pspec = NULL;

    pspec = g_param_spec_boolean ("window-open",
                                  "Window open prop",
                                  "Set window opening",
                                  FALSE,
                                  G_PARAM_CONSTRUCT | G_PARAM_READWRITE);

    /* ... */
}
```

Nous définissons donc le nom de notre propriété, sa valeur par défaut ainsi que différentes options (voir [GParamFlags](#) pour plus d'informations).

Ensuite nous installons notre propriété :

```
g_object_class_install_property (gobject_class,
                                PROP_WINDOW_OPEN,
                                pspec);
```

Pour en finir avec notre initialisation, nous devons spécifier deux fonctions qui serviront d'assesseurs :

```
gobject_class->set_property = vehicule_set_property;
gobject_class->get_property = vehicule_get_property;
```

Ces deux fonctions vont recevoir toutes les demandes d'accès ou de modifications de propriétés. Afin de traiter correctement chacune d'entre elles, nous effectuons un *switch/case* sur l'identifiant de la propriété :

```
static void vehicule_set_property (GObject *object, guint property_id,
                                  const GValue *value, GParamSpec *pspec)
{
    Vehicule *self = VEHICULE (object);
    VehiculePrivate *priv = NULL;

    priv = VEHICULE_GET_PRIVATE (self);
    switch (property_id)
    {
        case PROP_WINDOW_OPEN:
            priv->open_window = g_value_get_boolean (value);
            break;
        default:
            G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
            break;
    }
}

static void vehicule_get_property (GObject *object, guint property_id,
                                   GValue *value, GParamSpec *pspec)
{
    Vehicule *self = VEHICULE (object);
    VehiculePrivate *priv = NULL;

    priv = VEHICULE_GET_PRIVATE (self);
    switch (property_id)
    {
        case PROP_WINDOW_OPEN:
            g_value_set_boolean (value, priv->open_window);
            break;
        default:
            G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
            break;
    }
}
```

Afin de simplifier le code de notre exemple, les assesseurs sont de simples interfaces pour notre variable privée, cependant rien ne nous empêche de faire appel à d'autres fonctions (par exemple *personne_bouge_bras*, afin d'éviter tous problèmes lors de la fermeture des fenêtres...).

Pour en finir avec les propriétés, voici comment les modifier, soit une par une en passant pas les *GValue* :

```
GObject *vehicule = NULL;
GValue val = {0};

vehicule = g_object_new (TYPE_VEHICULE, NULL);
g_value_init (&val, G_TYPE_BOOLEAN);
g_value_set_boolean (&val, TRUE);
g_object_set_property (G_OBJECT (vehicule), "open-window", &val);
```

Soit par lot en passant directement la valeur :

```
g_object_set (vehicule, "open-vehicule", FALSE, NULL);
```

Syntaxe qui peut aussi être utilisée lors de la construction de l'objet si la propriété le permet :

```
vehicule = g_object_new (TYPE_VEHICULE, "open-vehicule", FALSE, NULL);
```

X - GObject Builder

GOB est un pré-processeur qui permet de créer facilement des classes dans un langage proche du C++ et de les convertir en code C.

Voici un exemple, extrêmement simple, de classe écrite pour GOB :

```
class Test:Object from G:Object
{
  public int i;
  private int j;
  protected int k;
}
```

Pour obtenir notre code source :

```
$ gob2 test.gob
```

Et nous obtenons trois fichiers portant le nom de notre classe :

test-object.h

```
/* Generated by GOB (v2.0.15) (do not edit directly) */

#include <glib.h>
#include <glib-object.h>
#ifndef __TEST_OBJECT_H__
#define __TEST_OBJECT_H__

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * Type checking and casting macros
 */
#define TEST_TYPE_OBJECT (test_object_get_type())
#define TEST_OBJECT(obj) G_TYPE_CHECK_INSTANCE_CAST((obj), test_object_get_type(), TestObject)
#define TEST_OBJECT_CONST(obj) G_TYPE_CHECK_INSTANCE_CAST((obj), test_object_get_type(), TestObject const)
#define TEST_OBJECT_CLASS(klass) G_TYPE_CHECK_CLASS_CAST((klass), test_object_get_type(), TestObjectClass)
#define TEST_IS_OBJECT(obj) G_TYPE_CHECK_INSTANCE_TYPE((obj), test_object_get_type ())

#define TEST_OBJECT_GET_CLASS(obj) G_TYPE_INSTANCE_GET_CLASS((obj), test_object_get_type(), TestObjectClass)

/* Private structure type */
typedef struct _TestObjectPrivate TestObjectPrivate;

/*
 * Main object structure
 */
#ifndef __TYPEDEF_TEST_OBJECT__
#define __TYPEDEF_TEST_OBJECT__
typedef struct _TestObject TestObject;
#endif
struct _TestObject {
  GObject __parent__;
  /*< public >*/
  int i;
  /*< private >*/
}
```

test-object.h

```
int k; /* protected */
TestObjectPrivate *_priv;
};

/*
 * Class definition
 */
typedef struct _TestObjectClass TestObjectClass;
struct _TestObjectClass {
    GObjectClass __parent__;
};

/*
 * Public methods
 */
GType test_object_get_type (void);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif
```

test-object-private.h

```
/* Generated by GOB (v2.0.15) (do not edit directly) */

#ifdef __TEST_OBJECT_PRIVATE_H__
#define __TEST_OBJECT_PRIVATE_H__

#include "test-object.h"

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

struct _TestObjectPrivate {
#line 4 "gobject-classe.gob"
    int j;
#line 16 "test-object-private.h"
};

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif
```

test-gobject.c

```
/* Generated by GOB (v2.0.15) (do not edit directly) */

/* End world hunger, donate to the World Food Programme, http://www.wfp.org */

#define GOB_VERSION_MAJOR 2
#define GOB_VERSION_MINOR 0
#define GOB_VERSION_PATCHLEVEL 15

#define selfp (self->_priv)

#include <string.h> /* memset() */

#include "test-object.h"

#include "test-object-private.h"
```

test-gobject.c

```
#ifndef G_LIKELY
#define __GOB_LIKELY(expr) G_LIKELY(expr)
#define __GOB_UNLIKELY(expr) G_UNLIKELY(expr)
#else /* ! G_LIKELY */
#define __GOB_LIKELY(expr) (expr)
#define __GOB_UNLIKELY(expr) (expr)
#endif /* G_LIKELY */
/* self casting macros */
#define SELF(x) TEST_OBJECT(x)
#define SELF_CONST(x) TEST_OBJECT_CONST(x)
#define IS_SELF(x) TEST_IS_OBJECT(x)
#define TYPE_SELF TEST_TYPE_OBJECT
#define SELF_CLASS(x) TEST_OBJECT_CLASS(x)

#define SELF_GET_CLASS(x) TEST_OBJECT_GET_CLASS(x)

/* self typedefs */
typedef TestObject Self;
typedef TestObjectClass SelfClass;

/* here are local prototypes */
static void test_object_init (TestObject * o) G_GNUC_UNUSED;
static void test_object_class_init (TestObjectClass * c) G_GNUC_UNUSED;

/* pointer to the class of our parent */
static GObjectClass *parent_class = NULL;

GType
test_object_get_type (void)
{
    static GType type = 0;

    if __GOB_UNLIKELY(type == 0) {
        static const GTypeInfo info = {
            sizeof (TestObjectClass),
            (GBaseInitFunc) NULL,
            (GBaseFinalizeFunc) NULL,
            (GClassInitFunc) test_object_class_init,
            (GClassFinalizeFunc) NULL,
            NULL /* class_data */,
            sizeof (TestObject),
            0 /* n_preallocs */,
            (GInstanceInitFunc) test_object_init,
            NULL
        };

        type = g_type_register_static (G_TYPE_OBJECT, "TestObject", &info, (GTypeFlags)0);
    }

    return type;
}

/* a macro for creating a new object of our type */
#define GET_NEW ((TestObject *)g_object_new(test_object_get_type(), NULL))

/* a function for creating a new object of our type */
#include <stdarg.h>
static TestObject * GET_NEW_VARG (const char *first, ...) G_GNUC_UNUSED;
static TestObject *
GET_NEW_VARG (const char *first, ...)
{
    TestObject *ret;
    va_list ap;
    va_start (ap, first);
    ret = (TestObject *)g_object_new_valist (test_object_get_type (), first, ap);
    va_end (ap);
    return ret;
}
```

test-gobject.c

```

}

static void
__finalize(GObject *obj_self)
{
#define __GOB_FUNCTION__ "Test:Object::finalize"
    TestObject *self G_GNUC_UNUSED = TEST_OBJECT (obj_self);
    gpointer priv G_GNUC_UNUSED = self->_priv;
    if(G_OBJECT_CLASS(parent_class)->finalize) \
        (* G_OBJECT_CLASS(parent_class)->finalize)(obj_self);
}
#undef __GOB_FUNCTION__

static void
test_object_init (TestObject * o G_GNUC_UNUSED)
{
#define __GOB_FUNCTION__ "Test:Object::init"
    o->_priv = G_TYPE_INSTANCE_GET_PRIVATE(o, TEST_TYPE_OBJECT, TestObjectPrivate);
}
#undef __GOB_FUNCTION__
static void
test_object_class_init (TestObjectClass * c G_GNUC_UNUSED)
{
#define __GOB_FUNCTION__ "Test:Object::class_init"
    GObjectClass *g_object_class G_GNUC_UNUSED = (GObjectClass*) c;

    g_type_class_add_private(c, sizeof(TestObjectPrivate));

    parent_class = g_type_class_ref (G_TYPE_OBJECT);

    g_object_class->finalize = __finalize;
}
#undef __GOB_FUNCTION__
    
```

Vous remarquerez que le code est relativement verbeux et surtout ne correspond pas forcément à votre style de codage, je pense en particulier aux constantes et noms de fonctions portant des noms réservés ou encore aux encombrants *extern "C"* qui pourraient être remplacés par les macro de la glib. Dans ce cas n'hésitez pas à modifier le code source de gob2 et en particulier le fichier *main.c* qui contient les fonctions permettant d'écrire les blocs de codes.

Par exemple pour cette histoire de *extern "C"*, voici les deux fonctions que j'ai modifié :

```

static void
print_header_prefixes(void)
{
    char *p;


    p = replace_sep(((Class *)class)->otype, '_');
    gob_strdup (p);
    out_printf(outh, "#ifndef H_%s#define H_%s\n\n", p, p);
    if(outph)
        out_printf(outph, "#ifndef H_%s_PRIVATE\n"
            "#define H_%s_PRIVATE\n"
            "#include \"%s.h\"\n\n", p, p, filebase);
    g_free(p);

    if( ! no_extern_c) {
        out_printf(outh, "G_BEGIN_DECLS\n\n");
        if(outph)
            out_printf(outph, "G_BEGIN_DECLS\n\n");
    }
}

static void
    
```

```
print_header_postfixes(void)
{
    char *p;

    p = replace_sep(((Class *)class)->otype, '_');
    gob_strup (p);
    if( ! no_extern_c)
        out_printf(outh, "\nG_END_DECLS\n");
    out_printf(outh, "\n#endif /* not H_%s */\n", p);
    if(outph) {
        if( ! no_extern_c)
            out_printf(outph, "\nG_END_DECLS\n");
        out_printf(outph, "\n#endif /* not H_%s_PRIVATE */\n", p);
    }
}
```

 *Si vous êtes sous Windows il est nécessaire de compiler gob2 à partir des sources. Cependant, vous risquez de rencontrer quelques problèmes.*

Une fois les sources décompressées, lancer le script configure, mais avant de lancer la commande make, ouvrez le fichier ./src/Makefile pour y apporter quelques modifications :

- Ligne 108, remplacez les deux lignes :

```
GLIB_CFLAGS = -Ic:/devel/target/559d8a8ed3bc64022b3b1de25a604527/include/glib-2.0
-Ic:/devel/target/559d8a8ed3bc64022b3b1de25a604527/lib/glib-2.0/include
GLIB_LIBS = -Lc:/devel/target/559d8a8ed3bc64022b3b1de25a604527/lib -lglib-2.0 -lintl
-liconv
```

Par :

```
GLIB_CFLAGS = -I/cygdrive/c/Program\ Files/CodeBlocks/include/glib-2.0
-I/cygdrive/c/Program\ Files/CodeBlocks/lib/glib-2.0/include
GLIB_LIBS = -L/cygdrive/c/Program\ Files/CodeBlocks/lib -lglib-2.0 -lintl -liconv -lfl
```

/cygdrive/c/Program\ Files/CodeBlocks/ étant bien sûr l'emplacement des différentes bibliothèques (glib 2.0 et flex).

- Ligne 200, remplacez :

```
bin_PROGRAMS = gob2
```

Par :

```
bin_PROGRAMS = gob2$(EXEEXT)
```

XI - Exemple complet

Pour illustrer mes propos j'ai utilisé une classe *Vehicule* tout au long de cet article, cependant on s'en sert rarement et elle est loin de montrer la puissance de GObject. Voici un exemple plus concret d'implémentation du **pattern singleton** :

singleton.h

```
#ifndef H_NJ_SINGLETON_28112007140332
#define H_NJ_SINGLETON_28112007140332

#include <glib-object.h>

#define TYPE_SINGLETON (singleton_get_type ())
#define SINGLETON(obj) (G_TYPE_CHECK_INSTANCE_CAST ((obj), TYPE_SINGLETON, Singleton))
#define SINGLETON_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST ((klass), TYPE_SINGLETON, SingletonClass))
#define IS_SINGLETON(obj) (G_TYPE_CHECK_INSTANCE_TYPE ((obj), TYPE_SINGLETON))
#define IS_SINGLETON_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass), TYPE_SINGLETON))
#define SINGLETON_GET_CLASS(obj) (G_TYPE_INSTANCE_GET_CLASS ((obj), TYPE_SINGLETON, SingletonClass))

typedef struct _Singleton Singleton;
typedef struct _SingletonClass SingletonClass;

struct _Singleton
{
    GObject parent;
};

struct _SingletonClass
{
    GObjectClass parent;

    /*< private >*/
    GObject *instance;
};

GType singleton_get_type (void);

#endif /* H_NJ_SINGLETON_28112007140332 */
```

singleton.c

```
#include "singleton.h"

static GObject *singleton_constructor (GType, guint, GObjectConstructParam *);
static void singleton_finalize (GObject *);

G_DEFINE_TYPE (Singleton, singleton, G_TYPE_OBJECT)

static void singleton_class_init (SingletonClass *klass)
{
    GObjectClass *gobject_class = G_OBJECT_CLASS (klass);

    gobject_class->constructor = singleton_constructor;
    gobject_class->finalize = singleton_finalize;
    klass->instance = NULL;
}

static void singleton_init (Singleton *self)
{
    g_return_if_fail (self != NULL);
}

static GObject *singleton_constructor (GType type, guint n_properties,
```

singleton.c

```
                                GObjectConstructParam *properties)
{
    SingletonClass *klass = NULL;

    klass = g_type_class_peek (TYPE_SINGLETON);
    if (klass->instance == NULL)
    {
        klass->instance = G_OBJECT_CLASS (singleton_parent_class)->constructor (type,
                                                                                   n_properties,
                                                                                   properties);
    }
    else
    {
        klass->instance = g_object_ref (klass->instance);
    }
    return klass->instance;
}

static void singleton_finalize (GObject *self)
{
    SingletonClass *klass = NULL;

    klass = g_type_class_peek (TYPE_SINGLETON);
    klass->instance = NULL;
    G_OBJECT_CLASS (singleton_parent_class)->finalize (self);
}
```

XII - Conclusion

Nous venons de voir une grande partie de la bibliothèque GObject mais ce n'est pas tout ce qu'il est possible de faire avec. Sachez qu'il est possible de créer des interfaces, des classes abstraites et des signaux.

Si vous voulez en savoir plus sur les rouages de la POO en C, je vous invite à lire ces quelques tutoriels en attendant la suite :

-  **Programmation orientée objets en C ? - La méthodologie** par CGI
-  **Programmation orientée objets en C ? - L'héritage** par CGI
-  **Object Oriented Programming in C** par Laurent Deniau

XIII - Remerciements

Merci à **Adrien Artero** par sa relecture orthographique de cet article.

