Université Savoie Mont Blanc,
UFR SceM
INFO504, L3 INFO

September 29, 2020
J.-O. Lachaud

Fiche 6 — Modules et compilation séparée, Tableaux multidimensionnels, fichiers, modifieurs de portée

---

Cette fiche rassemble différentes notions utiles lorsqu'on programme en C, et vous permet d'aller d'approfondir quelques aspects du langage. Ces éléments sont présentés sans ordre particulier.

1 Modules et compilation séparée

1.1 Fichiers en-tête ou ".h"

On distingue souvent en C les déclarations des fonctions de leurs définitions. La déclaration de la fonction s'écrit à l'aide du prototype de la fonction (valeur de retour nom(liste de paramètres)) terminé par ';'. La définition est le prototype plus le code source associé placé entre deux accolades '{' et '}'. Lorsque l'on veut appeler une fonction, le compilateur a besoin de connaître sa déclaration, mais pas forcément sa définition complète. C'est pourquoi on place en général les déclarations dans un fichier en-tête (.h) (avec les définitions des types) et les définitions dans un fichier source (.c). Du coup, on inclut juste le fichier en-tête dans un autre programme si on veut utiliser des fonctions définies dans un autre fichier source.

pile.h

```c
#ifndef _PILE_H_
#define _PILE_H_
struct SPile {
  int elems[ 100 ]; // stockera les éléments
  int nb; // nb elements
};
typedef struct SPile Pile;

void Pile_init( Pile* p );
int Pile_vide( Pile* p );
int Pile_pleine( Pile* p );
void Pile_push( Pile* p, int e );
int Pile_sommet( Pile* p );
void Pile_pop( Pile* p );

#endif /* ifndef _PILE_H_ */
```

pile.c

```c
#include <stdio.h>
#include "pile.h"

void Pile_init( Pile* p )
{
  p->nb = 0;
}

int Pile_vide( Pile* p )
{
  return p->nb == 0;
}

...
```

fibonacci.c

```c
#include <stdio.h>
#include "pile.h"

/* Fibonacci */
int main( int argc, char* argv[] )
{
  Pile P;
  Pile_init( &P );
  int val = 10;
  if ( argc > 1 )
    val = atoi( argv[ 1 ] );
  Pile_push( &P, val );
  int somme = 0;
  while ( ! Pile_vide( &P ) )
    {
      int v = Pile_sommet( &P );
      Pile_pop( &P );
      if ( v <= 1 ) somme += v ;
      else
        {
          Pile_push( &P, v-1 );
          Pile_push( &P, v-2 );
        }
    }
  printf( "Fibonacci( %d ) = %d \n", val, somme );
  return 0;
}
```

On compilera tout cela ainsi (en une ligne) (en fait compilation des deux fichiers .c, puis éditions des liens entre) :

```
$ gcc fibonacci.c pile.c -o fibonacci
```

Les puristes feront :

```
$ gcc -c pile.c      # compile pile.c
$ gcc -c fibonacci.c # compile fibonacci.c
$ gcc fibonacci.o pile.o -o fibonacci # édition des liens
```

1.2 La compilation (gcc -c)

Pour accélérer la fabrication d'un programme et pour mieux partager le code entre programmes, on sépare l'étape de compilation de l'étape de l'édition des liens en C (ou C++).

En réalité, la compilation d'un module C (source.c) vérifie que le code est bien du C, puis génère le code assembleur correspondant aux fonctions définies dans le source. Le tout est stocké dans un fichier source.o appelé *fichier objet*. Les fonctions extérieures utilisées par ce module sont juste indiquées comme manquantes et ne seront reliées au programme qu'à l'édition des liens qui met tout ensemble. La commande nm permet de lister les symboles utilisés par un programme ou un fichier objet. Sur l'exemple de la pile et fibonacci précédent, voilà ce que cela donne:

```
[you] nm pile.o
0000000000000000 T Pile_init
0000000000000034 T Pile_pleine
00000000000000ed T Pile_pop
0000000000000051 T Pile_push
00000000000000a8 T Pile_sommet
0000000000000018 T Pile_vide
                 U puts
[you] nm fibonacci.o
                 U Pile_init
                 U Pile_pop
                 U Pile_push
                 U Pile_sommet
```

```
                 U Pile_vide
                 U atoi
0000000000000000 T main
                 U printf
```

Le résultat sur le programme `fibonacci` est plus complexe, mais est essentiellement la fusion de tous les symboles de chaque module, en vérifiant que pour chaque symbole indéfini, ce symbole est défini dans un autre module. Ici, 'T' dit que la fonction est définie, 'U' dit que la fonction est indéfinie. Comme on peut voir, seules les fonctions `puts` et `printf` restent indéfinies lorsqu'on mettra ensemble ces deux modules.

1.3 L'édition des liens (ld ou plus simplement gcc)

Lorsqu'on veut rassembler plusieurs fichiers objets ensembles pour faire un programme, on utilise la commande `gcc` tout court, qui est un raccourci pour faire l'édition des liens d'un programme C, en rajoutant en même temps la bibliothèque C standard.

Sur l'exemple précédent, on écrirait donc:

```
[you] gcc pile.o fibonacci.o -o fibonacci
```

Ce qui crée l'exécutable `fibonacci`.

Une *bibliothèque de fonctions* est un ensemble de fichier objets rassemblés sous un seul nom. Par exemple `libc` est la bibliothèque C standard et contient les codes assembleur de `printf`, `atoi`, etc.

Si vous avez besoin d'autres bibliothèques où des fonctions sont définies, il faut les rajouter sur la ligne d'éditions des liens. Par exemple, si vous utilisez une fonction mathématique (de math.h), celles-ci ne sont pas dans la bibliothèque C standard (`libc`), mais dans la bibliothèque mathématique `libm`. Si `libm` est installé dans un chemin système habituel (`/usr/lib/`), alors on ferait l'édition des liens ainsi:

```
[you] gcc module1.o module2.o ... -lm -o mon_programme
```

De façon plus générale, lorsque vous utilisez une bibliothèque de fonctions installée à un endroit un peu exotique (par exemple, dans votre `${HOME}/local` si vous n'avez pas les droits systèmes), voilà ce qu'il faut faire. On suppose que vous voulez utiliser la bibliothèque GMP de calcul avec des nombres arbitrairement grands. Le fichier en-tête est "`gmp.h`". Il est logiquement mis dans `${HOME}/local/include`. La bibliothèque est "`libgmp`". Elle est logiquement mise dans `${HOME}/local/lib`. Votre programme "`essai.c`" ressemblera à:

```c
/* include systeme */
#include <stdio.h>

/* include autre bibliothèques */
#include "gmp.h"

/* vos includes de vos modules à vous. */
#include "autre_module.h"

/* votre programme */
...
int main( ... )
...
```

On le compilerait ainsi (on note le `-I` pour définir les chemins où chercher les fichiers en-tête) :

```
[you] gcc -I${HOME}/local/include -c essai.c
[you] gcc -I${HOME}/local/include -c autre_module.c
```

Et l'édition des liens se ferait ainsi (on note le `-L` pour définir les chemins où chercher les bibliothèques, et le `-lnom` pour indiquer le nom de la bibliothèque) :

```
[you] gcc -L${HOME}/local/lib essai.o autre_module.o -lgmp -o essai
```

1.4 Encapsulation des données

Le C est un langage qui donne extrêmement de liberté au programmeur. Il n'empêche que cette liberté induit une discipline de programmation. Il est important de la respecter au mieux afin d'avoir des programmes réutilisables ou qui "passent à l'échelle". Ainsi, les champs d'une structure sont accessibles à tout endroit d'un programme. Il est néanmoins préferrable de définir un ensemble de fonctions pour manipuler la structure, et de passer par ces fonctions. Cette approche est raisonnable car:

1. quand on crée une structure, on a en tête un certain usage (une liste de services) que l'on cherche à regrouper sous un seul nom. On ne cherche pas à faire plusieurs usages distincts de la même structure.

2. on ne se préoccupe pas vraiment de comment seront gérés ces services dans cette structure lorsque l'on s'en sert ailleurs.

3. parfois, le programmeur veut faire évoluer l'implémentation sans toucher aux services rendus. Si l'utilisateur n'a pas utilisé directement des champs internes de la structure mais n'utilise que des fonctions de manipulation, tout devrait être donc transparent pour lui.

C'est le principe de l'*encapsulation*, bien connue des programmeurs objet. L'exemple ci-dessus de la Pile vous montre comment cela est réalisé en C. On ne peut pas bloquer l'utilisation directe des champs de la pile, mais c'est déconseillé. On veut que l'utilisateur passe par les fonctions données. Du coup, les fonctions associées à une structure T ont en général comme premier paramètre un pointeur vers T, afin d'agir dessus.

Une convention raisonnable est que les fonctions soient préfixées par le nom de la structure. De plus, on associe des fonctions de création/destruction, similairement à des constructeurs/destructeurs des langages à objet:

```
void T_init( T* t, ... ); /* construction spécifique de t.  */
void T_termine( T* t );   /* destruction spécifique de t. */
```

Attention, les fonctions précédentes ne s'occupent pas de l'allocation/désallocation éventuelle de la variable pointée par `t`, mais seulement de l'initialisation/terminaison des champs de la structure. On peut associer d'autres fonctions pour le faire:

```
T* T_creer();              /* allocation dynamique d'une structure de */
                           /* type T, appelle init. */
void T_detruire( T* t );   /* appelle termine, t n'est plus une zone */
                           /* mémoire valide après */
```

Sur l'exemple de la pile, voilà comment on peut écrire tout cela:

```
Pile* Pile_creer()
{
  Pile* t = (Pile*) malloc( sizeof( Pile ) );
  Pile_init( t );
  return t;
}
void Pile_init( Pile* p )
{
  p->nb = 0;
}
voir Pile_termine( Pile* p );
{
  p->nb = 0;
}
void Pile_detruire( Pile* t )
{
  Pile_termine( t );
  free( t );
}
...
{
  /* utilisation sans allocation dynamique. */
  Pile P0;
  Pile_init( &P0 );
  Pile_push( &P0, 10 );
```

```
  ...
  Pile_termine( &P0 );

  /* utilisation avec allocation dynamique. */
  Pile * P1 =  Pile_creer();
  Pile_push( P1, 10 );
  ...
  Pile_detruire( P1 );
}
```

Plus utile, voilà comment s'écrirait une pile extensible.

```
struct SPile {
  int* elems; // stockera les éléments
  int nb;  // nb elements
  int max; // nb de cases allouees.
};
typedef struct SPile Pile;

void Pile_init( Pile* p )
{
  p->max = 100;
  p->elems = (int*) malloc( p->max * sizeof( int ) );
  p->nb = 0;
}
Pile* Pile_creer()
{
  Pile* p = (Pile*) malloc( sizeof( Pile ) );
  Pile_init( p );
  return p;
}
void Pile_termine( Pile* p )
{
  p->max = 0;
  p->nb = 0;
  free( p->elems );
}
void Pile_detruire( Pile* p )
{
  Pile_termine( p );
  free( p );
}

void Pile_agrandir( Pile* p )
{
  p->elems = (int*) realloc( p->elems, 2 * p->max * sizeof( int ) );
  p->max *= 2;
}

int Pile_pleine( Pile* p )
{
  return p->nb == p->max;
}

/* le reste est inchange. */
```

On voit qu'on se rapproche de la notion d'objet. En C++, on écrirait `p.pleine()` sur un objet `p` de type `Pile`. En, on écrit `Pile_pleine( p )` où `p` est un pointeur de `Pile`. Malheureusement, contrairement à JAVA ou C++, il est un peu plus difficile de faire l'héritage ou le polymorphisme (fonctions virtuelles).

2 Tableaux à plusieurs dimensions

Le C incorpore un mécanisme pour déclarer des tableaux à plusieurs dimensions de taille donnée. En fait, il s'agit d'un tableau monodimensionnel, mais dont le premier indice saute autant de cases que spécifiées pour aller à la ligne suivante (10 dans l'exemple ci-dessous, car chaque ligne a 10 colonnes). Les tableaux à plusieurs dimensions sont bien pratiques, mais imposent une taille fixe connue à l'instanciation, ce qui limite leur usage à des cas particuliers (type jeux, sudoku, etc), et les

empêche d'être utiliser en traitement d'image (où la taille n'est pas anticipable).

```c
#include <stdio.h>

int matrice[ 8 ][ 10 ]; // 8 lignes, 10 colonnes

int main( void )
{
  printf( "sizeof(int) = %ld\n", sizeof(int) );
  printf( "matrice = %p\n", matrice );
  printf( "&matrice = %p\n", &matrice );
  printf( "&matrice[0] = %p\n", &matrice[0] );
  printf( "&matrice[0][0] = %p\n", &matrice[0][0] );
  printf( "&matrice[0][1] = %p\n", &matrice[0][1] );
  printf( "&matrice[0][2] = %p\n", &matrice[0][2] );
  printf( "&matrice[0][9] = %p\n", &matrice[0][9] );
  printf( "&matrice[1] = %p\n", &matrice[1] );
  printf( "&matrice[1][0] = %p\n", &matrice[1][0] );
  printf( "&matrice[1][1] = %p\n", &matrice[1][1] );
  return 0;
}
```

Ce programme s'exécute ainsi :

```
sizeof(int) = 4
matrice = 0x601040
&matrice = 0x601040
&matrice[0] = 0x601040
&matrice[0][0] = 0x601040
&matrice[0][1] = 0x601044
&matrice[0][2] = 0x601048
&matrice[0][9] = 0x601064
&matrice[1] = 0x601068
&matrice[1][0] = 0x601068
&matrice[1][1] = 0x60106c
```

3 Les fichiers en C

Il y a plusieurs façons de faire. Plutôt bas niveau (en gros UNIX/linux), ce sont les fonctions `open`, `creat`, `lseek`, `read`, `write`,`close`. L'idée est de créer un descripteur de fichier (un index/handle qui caractérise le fichier), puis des fonctions qui lisent ou écrivent des octets, ou encore sautent des octets dans le fichier. Pour avoir plus d'informations, chercher les manual pages, e.g. `man 2 open`. Si on peut les éviter, c'est mieux car ces fonctions se conforment à SVr4, BSD ou POSIX, mais ne sont pas cross-systèmes.

Plus haut niveau, ce sont les fonctions `fopen`, `fread`, `fwrite`, `fprintf`, `fscanf`, `feof`, `fclose`, `fseek`, `ftell`, `frewind`, qui elles font partie des standards C89 et C99. On crée une structure FILE@*@ qui permet une manipulation assez haut niveau du fichier, un peu sous forme d'un flux (en entrée ou en sortie).

```c
/* Ecrit 10 lignes dans un fichier dont le nom est donné en paramètre
   sur la ligne de commande. */
#include <stdio.h>

int main( int argc, char* argv[] )
{
  if ( argc != 2 ) {
    fprintf( stderr, "Usage: %s <fname>\n", argv[ 0 ] );
    return 1;
  }
  char* fname = argv[ 1 ];
  FILE* f = fopen( fname, "a" ); /* Ouvert en écriture / append. */
  if ( f == NULL ) {
    fprintf( stderr, "Erreur en écriture sur %s.\n", argv[1] );
    return 1;
  }
```

```
    for ( int i = 0; i < 10; i++ )
      fprintf( f, "Ceci est la ligne %d.\n", i );
    fclose( f );
    return 0;
}
```

4 Les modifieurs de portée: `extern`, `static`, `auto`, `register`

Chaque fois qu'un symbole est défini en C, il tombe dans une des quatres classes de stockage du C (storage class en anglais). Souvent, l'utilisateur n'a pas besoin de le préciser car il y a toujours une classe de stockage par défaut. Pour les variables :

`auto` Indique que la variable est locale et sera automatiquement allouée/désallouée sur la pile d'exécution. La variable est dite automatique. C'est la classe par défaut. Ce mot-clé est très rarement utilisé, même si en fait, 99% des variables sont automatiques.

```
void fct( int c )
{
  auto int a = 0;
  printf("&a = %p\n", &a ); /* Vous etes sur que a est sur la pile */
  ...
}
```

`register` C'est un indice pour le compilateur que vous souhaitez que cette variable soit prioritairement un registre du processeur. Sans doute, vous pensez que c'est *la* variable à optimiser. Pas de garantie néanmoins. La seule différence est que vous n'avez plus le droit d'utiliser l'opérateur d'adresse "&" sur cette variable. Très peu utilisé aussi.

`static` Spécifie que la variable n'est pas allouée sur la pile mais dans un segment de donnée spécifique à ce fichier en cours de compilation. La valeur de la variable n'est donc pas perdue si on quitte la fonction où la variable a été définie. `static` peut aussi être utilisée pour spécifier une variable globale à un fichier. Attention, si `static` est utilisée sur une variable dans un fichier entête partagé par plusieurs fichiers, la variable est dupliquée autant de fois que nécessaire (si `static_titi_h_var` ci-dessous).

`extern` Spécifie que la variable n'est pas allouée sur la pile mais dans un autre fichier objet. Il faudra donc qu'un autre module le définisse ailleurs.

Pour les fonctions :

`static` Spécifie que la fonction est spécifique à ce fichier en cours de compilation, et ne sera donc pas disponible pour d'autres modules. Le symbole n'est pas exporté par le module.

`extern` Spécifie que la fonction est définie ailleurs. Il faudra donc qu'un autre module le définisse. C'est le comportement par défaut des fonctions déclarées. Si on reprend l'exemple de la pile, toutes les fonctions "publiques" sont mises dans le fichier entête `Pile.h` (on devrait mettre `extern` devant chaque fonction, mais c'est fait par défaut. Puis les fonctions sont définies dans le fichier `Pile.c`. Si on ne souhaite absolument exporter certaines fonctions, on met *static* devant ces fonctions.

Les bouts de code ci-dessous montrent les différentes possibilités de `extern` et `static`.

toto-1.c

```
#include <stdio.h>
#include "titi-1.h"

int toto_c_var;
static int static_toto_c_var;

int main()
{
  toto_c_var = 0;
  static_toto_c_var = 1;
  extern_titi_h_var = 3;
  titi_h_var = 3;
  static_titi_h_var = 3;
  fct();
  extern_fct();
  printf( "extern_titi_h_var = %d\n", extern_titi_h_var );
  printf( "titi_h_var = %d\n", titi_h_var );
  printf( "static_titi_h_var = %d\n", static_titi_h_var );
  return 0;
}
```

titi-1.h

```
#ifndef _TITI_1_H_
#define _TITI_1_H_

static int static_titi_h_var; /* a eviter en general. */
int titi_h_var;               /* extern par default */
extern int extern_titi_h_var;

void fct();                   /* extern par default */
extern void extern_fct();

#endif /* _TITI_1_H_*/
```

titi-1.c

```
#include "titi-1.h"

static int static_titi_c_var;
int titi_c_var;
int extern_titi_h_var;

void fct()
{
  extern_titi_h_var = 4;
  titi_h_var = 4;
  static_titi_h_var = 4;
}
void extern_fct()
{
  extern_titi_h_var = 5;
  titi_h_var = 5;
  static_titi_h_var = 5;
}
```

nm toto-1.o

```
                 U extern_fct
                 U extern_titi_h_var
                 U fct
0000000000000000 T main
                 U printf
0000000000000000 b static_titi_h_var
0000000000000004 b static_toto_c_var
0000000000000004 C titi_h_var
0000000000000004 C toto_c_var
```

nm titi-1.o

```
0000000000000024 T extern_fct
0000000000000004 C extern_titi_h_var
0000000000000000 T fct
0000000000000004 b static_titi_c_var
0000000000000000 b static_titi_h_var
0000000000000004 C titi_c_var
0000000000000004 C titi_h_var
```

execution

```
[you] gcc toto-1.o titi-1.o -o prog
[you] ./prog
extern_titi_h_var = 5
titi_h_var = 5
static_titi_h_var = 3
```

5 Programmation orientée objet en C ?

Même si le langage n'est certainement pas le plus adapté pour faire cela, on peut recréer des comportements typiques de programmes objet en C. Nous listons quelques manières de faire ci-dessous.

5.1 Héritage simple

Il est très fréquent en langage objet de vouloir créer une classe B sous-classe d'une classe A. Ensuite, on souhaite se servir d'une variable de type B ou de type A indifféremment. Pour ce faire, on utilise le fait que le C garantit que les champs sont alloués de manière consécutives et toujours dans l'ordre de lecture. Du coup, l'écriture ci-dessous permet d'émuler une relation d'héritage entre `Etudiant` et `Personne`:

```
typedef struct {
  char* nom;
  char* prenom;
  int age;
} Personne;

typedef struct {
  Personne base;
  char* numero_etd;
  char* filiere;
} Etudiant;
```

```c
void affiche( Personne* p )
{
  printf( "%s %s %d\n", p->nom, p->prenom, p->age );
}

void affiche_etd( Etudiant* p )
{
  printf( "%s %s %d %s %s\n",
          p->base->nom, p->base->prenom, p->base->age,
          p->numero_etd, p->filiere );
}

int main( void )
{
  Personne P1 = { "Bon", "Jean", 20 };
  Etudiant P2 = { "Canthrope", "Emilie", 22, "0123456A", "L1 Math" };
  affiche( &P1 );
  affiche( (Personne*) &P2 );
  affiche_etd( &P2 );
  affiche_etd( (Etudiant*)&P1 ); /* Incorrect */
  return 0;
}
```

L'exemple précédent donne à l'exécution:

```
Bon Jean 20
Canthrope Emilie 22
Canthrope Emilie 22 0123456A L1 Math
Bon Jean 20 (null) (null)
```

Dit autrement, le début de la structure `Etudiant` est le même que la structure `Personne`. On peut donc les substituer à l'exécution, car elles tombent au même endroit mémoire.

5.2 Le polymorphisme

On voudrait que les structures puissent être manipulée de la même façon tout en ayant des comportements différents à l'exécution dans certains cas. on utilise alors les pointeurs de fonctions. Prenons l'exemple ci-dessous:

```c
#include <stdio.h>

typedef const char* (*FctCri)();

typedef struct {
  FctCri cri;
} Animal;

const char* rugir()
{
  return "Groarrrr !";
}

const char* miauler()
{
  return "Miaou...";
}

const char* aboyer()
{
  return "Wooaah !";
}

void Lion_init( Animal* a )
{
  a->cri = rugir;
}

void Chat_init( Animal* a )
{
```

```
  a->cri = miauler;
}

void Chien_init( Animal* a )
{
  a->cri = aboyer;
}

int main()
{
  Animal t[ 5 ];
  Lion_init( t + 0 );
  Chat_init( t + 1 );
  Chien_init( t + 2 );
  Lion_init( t + 3 );
  Chien_init( t + 4 );
  for ( int i = 0; i < 5; ++i )
    printf( "cri de t[ %d ] : %s\n", i, t[ i ].cri() );
  return 0;
}
```

On voit que l'on change la fonction cri (qui devient une sorte de méthode de la structure Animal) à l'initialisation. L'utilisation des pointeurs de fonction est donc équivalent aux appels de méthode des langages C++ ou JAVA. On peut même changer dynamiquement la fonction appelée en lui reaffectant une autre valeur. GTK utilise ce mécanisme pour gérer quelle est la fonction appelée en réaction à un événement.

L'exemple ci-dessous est un peu plus complet. On définit une fonction générique à tout animal, plus des fonctions spécialisés pour différents animaux. Attention, `Lion`, `Chat` et `Chien` ne sont pas vraiment des sous-classes de `Animal`: ils partagent tous les données de `Animal` (juste le prénom donc). On note l'utilisation de MACROs pour alléger (un peu) les notations et améliorer la visibilité.

```
/* Polymorphisme-3.c */
#include <stdio.h>

#define FCT( __ret, __nom, __params ) \
  __ret (* __nom ) __params
#define DEFFCT( __ret, __nom, __params ) \
  __ret __nom __params
#define BINDFCT( __a, __nom, __fct ) \
  __a->__nom = __fct


typedef struct SAnimal {
  const char* petit_nom;
  FCT( const char*, nom, () );
  FCT( const char*, cri, () );
  FCT( void, quiSuisJe, ( struct SAnimal* a ) );
} Animal;

DEFFCT( void, Animal_quiSuisJe, ( Animal* a ) )
{
  printf( "Je suis %s le %s. Je fais %s\n",
          a->petit_nom, a->nom(), a->cri() );
}

DEFFCT( const char*, rugir, () )
{ return "Groarrrr !"; }
DEFFCT( const char*, miauler, () )
{ return "Miaou..."; }
DEFFCT( const char*, aboyer, () )
{ return "Wooaah !"; }
DEFFCT( const char*, lion, () )
{ return "lion"; }
DEFFCT( const char*, chat, () )
{ return "chat"; }
DEFFCT( const char*, chien, () )
{ return "chien"; }

void Animal_init( Animal* a, const char* prenom )
```

```
{
  BINDFCT( a, quiSuisJe, Animal_quiSuisJe );
  a->petit_nom = prenom;
}

void Lion_init( Animal* a, const char* prenom )
{
  Animal_init( a, prenom );
  BINDFCT( a, nom, lion );
  BINDFCT( a, cri, rugir );
}

void Chat_init( Animal* a, const char* prenom )
{
  Animal_init( a, prenom );
  BINDFCT( a, nom, chat );
  BINDFCT( a, cri, miauler );
}

void Chien_init( Animal* a, const char* prenom )
{
  Animal_init( a, prenom );
  BINDFCT( a, nom, chien );
  BINDFCT( a, cri, aboyer );
}

int main()
{
  Animal t[ 5 ];
  /* Tous ces animaux sont polymorphes ! */
  Lion_init( t + 0, "Mufasa" );
  Chat_init( t + 1, "Le Chat Botté" );
  Chien_init( t + 2, "Médor" );
  Lion_init( t + 3, "Richard" );
  Chien_init( t + 4, "Rex" );
  for ( int i = 0; i < 5; ++i )
    t[ i ].quiSuisJe( t + i );
  return 0;
}
```

Ce qui affiche:

```
Je suis Mufasa le lion. Je fais Groarrrr !
Je suis Le Chat Botté le chat. Je fais Miaou...
Je suis Médor le chien. Je fais Wooaah !
Je suis Richard le lion. Je fais Groarrrr !
Je suis Rex le chien. Je fais Wooaah !
```

On note une curiosité : la case `t+i` apparaît deux fois dans un appel de méthode ! En fait, le premier `t+i` (ici `t[i]`) définit quelle est la fonction spécifique appelée, tandis que le deuxième définit quel est l'objet sur lequel cette fonction est appelée. En POO classique, les deux sont toujours identiques. On écrirait donc plutôt quelque chose du genre :

```
#define CALL0( __ptr, __fct ) \
  (__ptr)->__fct( __ptr )
#define CALL1( __ptr, __fct, __param1 ) \
  (__ptr)->__fct( __ptr, __param1 )
#define CALL2( __ptr, __fct, __param1, __param2 ) \
  (__ptr)->__fct( __ptr, __param1, __param2 )
...
  for ( int i = 0; i < 5; ++i )
    CALL0( t + i, quiSuisJe );
...
```

Depuis C99, il est aussi possible d'utiliser des macros avec nombre variable d'argument. Nous l'utilisons ainsi (extension de gcc) pour faire une seule macro CALL pour appeler avec une fonction avec un nombre quelconque d'arguments.

```
/* Extension de gcc pour utiliser la macro sans argument */
#define CALL( __ptr, __fct, ...) \
```

```
  (__ptr)->__fct( __ptr, ##__VA_ARGS__ )
...
  for ( int i = 0; i < 5; ++i )
    CALL( t + i, quiSuisJe );
...
```

Comme on le voit, on peut faire de la POO en C, même s'il faut beaucoup plus de discipline que dans d'autres langages spécialisés. Il est évident que vous devez prendre en compte les spécificités d'un langage lorsque vous le choisissez pour votre problématique. Si vous ne recherchez pas spécialement la performance et que vous voulez faire du tout objet avec un usage massif du polymorphisme, il est sans (aucun) doute préferable de choisir un autre langage que le C.

A noter néanmoins que les différentes méthodes présentées ci-dessus vous montrent comment les programmes objets (même non C) sont traduits en code machine. Les pointeurs de fonction sont des tables virtuelles (en C++, JAVA) qui mémorisent la fonction à appeler à l'exécution. L'objet lui-même (this) est bien un pointeur vers une structure de donnée. Chaque fois qu'un objet est créé ou détruit, constructeurs et destructeurs sont simplement appelés automatiquement. Vous pouvez donc cerner à la fois le gain en modélisation de la POO, mais aussi son léger surcoût en terme d'exécution (indirection supplémentaire à chaque appel de méthode virtuelle).