Notes de cours INFO504, L3 STIC INFO
Programmation C

Jacques-Olivier Lachaud
LAMA, Université de Savoie
http://www.lama.univ-savoie.fr/wiki

4 septembre 2019

Introduction

Ce document trace les grandes lignes d'un cours de programmation C. Ce cours s'adresse à des étudiants de niveau intermédiaire en informatique, ayant déjà programmé avec un autre langage de programmation impératif. L'objectif est de connaître les spécificités du langage C, et de savoir programmer avec ce langage à la fois à bas niveau et à haut niveau. Pour la syntaxe du langage C, *The C programming Langage, Kernighan/Ritchie* est très bon, mais n'intègre pas les modifications de la dernière version du C. On peut conseiller aussi le *Méthodologie de la programmation en C, Braquelaire, Ed. Dunod, 2005*, qui intègre la norme C99. Vous pouvez bien sûr consulter le *wikilivre sur la programmation C,* `fr.wikipedia.org`.

Ce document ne donnera pas tous les éléments de la syntaxe du C, loin de là. Il existe des ressources disponibles qui le font (cf. plus haut).

Plan

1. Introduction au langage C
   (a) Pourquoi le langage C
   (b) Historique
   (c) Caractéristiques importantes
   (d) Cycle de développement d'un programme C
   (e) Eléments de base du langage
   (f) Expressions
   (g) Affectations
   (h) Instructions conditionnelles
   (i) Structures répétitives
   (j) Entrées-sorties simples
   (k) Le constructeur `struct`
   (l) Exercices
2. Fonctions, passage de paramètres, pointeurs
   (a) Définitions des fonctions
   (b) Fonctions récursives
   (c) Passage de paramètres
   (d) Passage par adresse
   (e) Les pointeurs
   (f) Les tableaux sont (presque toujours) des pointeurs
   (g) Le pointeur `void*`
   (h) Arithmétique des pointeurs
   (i) Allocation dynamique
   (j) Pointeurs sur les structures
   (k) Un exemple complet : les piles en C
   (l) La pile d'exécution du processus
   (m) Tableaux comme si ce n'étaient pas des pointeurs
   (n) Tableaux à plusieurs dimensions
   (o) Structures auto-référentes
   (p) Exercices
3. Usages et pratiques de la programmation en C
   (a) Fichiers en-tête ou ".h"

(b) Le préprocesseur
(c) La compilation (gcc -c)
(d) L'édition des liens (ld ou plus simplement gcc)
(e) Encapsulation des données
(f) Programmation orientée objet en C ?
    i. L'héritage simple
    ii. Le polymorphisme
(g) Les fichiers en C
(h) Les modifieurs de portée : `extern`, `static`, `auto`, `register`
(i) Autres spécificités du C

1 Introduction au langage C

1.1 Pourquoi le langage C

C'est le langage pour la programmation dite bas-niveau, c'est-à-dire écrire les systèmes d'exploitation, les drivers de périphériques, du code embarqué, mais aussi pour écrire les machines d'exécution des langages interprétés. Par exemple, Unix, les différentes versions de Window, Mac OS X, GNU/Linux ont été écrites en C. Les interpréteurs de Python, Ruby, PHP, Perl sont en C. Votre shell est écrit en C.

Beaucoup de langage suivent l'esprit du langage C. Connaître le C, c'est un peu connaître leurs racines communes. Le C, quoique ayant une syntaxe tout à fait lisible, est très proche de la programmation sur microprocesseurs, c'est-à-dire du langage machine. Le code écrit en C est quasiment aussi rapide que du code écrit en assembleur directement (et même souvent plus rapide sur les processurs RISC), est beaucoup plus lisible, mais surtout ce code est portable.

Les compilateurs C (notamment GCC le plus connu et le plus versatile) permettent d'écrire des programmes qui vont s'exécuter sur quasiment n'importe quelle machine, processeur ou architecture existant. Ils permettent par exemple d'écrire les systèmes des nouveaux processeurs.

Quoique le langage C présente peu de caractéristiques des langages haut-niveau, il permet néanmoins aussi la programmation haut-niveau. L'interface de Gnome, GTK, est complètement écrite en C. Néanmoins, il faut reconnaître que cela demande au programmeur un peu plus de discipline de programmation.

Beaucoup de langages se sont inspirés de C : C++, C#, Objective C, JAVA notamment. Une bonne compréhension du C est indispensable pour bien maîtriser le langage C++. En fait comprendre le C, c'est comprendre comment fonctionne le processeur, et donc comment s'exécutent tous les programmes sur un ordinateur. On peut par exemple facilement déterminer la complexité d'un programme en C, car la plupart des instructions C sont de complexité constante (i.e. prenne un temps constant).

1.2 Historique

Inventé par Ken Thompson et Denis Ritchie en 1972 (progressivement en fait, d'abord B), pour écrire un UNIX plus portable. C'est le C originel ou K & R.

Standardisé en 1989, il devient le ANSI C. Il rationalise quelques éléments de syntaxe bizarres du premier C. Maintenant, la norme est le C99, essentiellement *backward-compatible* avec quelques nouvelles caractéristiques (tableaux dynamiques notamment).

1.3 Caractéristiques importantes

Les caractéristiques principales du langage C sont :
— langage compilé
— portée limitée des variables
— programmation impérative : opérations de calcul, conditionnelles, structures répétitives
— programmation structurée : fonctions, modules, calculs récursifs

— structure de données aussi évoluées que souhaitées par agrégation, tableaux, union.
— typage en partie faible à la compilation
— accès à la mémoire via des pointeurs
— une partie de la gestion de la mémoire par l'utilisateur (allocation dynamique)
— pointeurs de fonction pour éventuel polymorphisme.
— préprocesseur pour limiter les temps de compilation et favoriser la compilation séparée
— bibliothèque standard (libc) : chaînes de caractères, I/O, mathématiques
— syntaxe concise : beaucoup d'opérateurs, blocs par accolades, fin d'instruction par ;

Ce qui manque ?

— pas d'assignation directe de tableaux (mais assignation de structures)
— pas de gestion automatique de la mémoire allouée, pas de ramasse-miettes
— pas de test de mauvais indice pour un tableau
— pas de gestion d'exceptions
— pas de surcharge d'opérateurs ou de fonctions
— pas de support pour faciliter la programmation objet (héritage, polymorphisme)
— pas de bibliothèque native pour le multithread ou le réseau, ni même pour l'affichage graphique (juste console).

Du coup, POSIX a établi des normes pour certaines de ces bibliothèques (e.g., `libpthread`). Sinon, les systèmes UNIX, Linux ou Windows fournissent leurs propres bibliothèques, plus ou moins portables.

1.4 Cycle de développement d'un programme C

Le code C est écrit dans des fichiers texte, appelés *sources*, terminés par le suffixe `.c`. Un programme, appelé *compilateur C*, permet de traduire ces sources en un programme exécutable directement sur une architecture choisie. Il existe de nombreux compilateurs : gcc (pour GNU CC), icc (Intel), Visual studio, Borland C.

En réalité, la compilation se fait en plusieurs étapes : le préprocesseur, la compilation (traduction en langage machine), et l'édition des liens (connexions des différents modules compilés séparément).

Voilà l'exemple classique Hello World, placé dans le fichier `hello.c` :

```
#include <stdio.h>

int main( void )
{
  printf( "Hello the World !\n" );
  return 0;
}
```

On écrira pour compiler ce fichier sur votre shell :

```
you@machine$ gcc -c hello.c
```

Cela vous rend la main sans rien dire si le programme s'est compilé sans problème. Normalement, un fichier `hello.o` a été créé. Il contient du code machine binaire parfaitement illisible pour vous. Néanmoins, vous ne pouvez pas l'exécuter. En effet, votre code a bien été compilé (en environ 500 octets de code machine), mais il lui manque le lien avec toutes les fonctions écrites dans le système que vous appelez dans votre code. Ici, vous utilisez la fonction (plutôt complexe) `printf`. Pour faire l'édition des liens avec la bibliothèque système C. On écrira :

```
you@machine$ gcc hello.o -o hello
```

Cela fabrique un exécutable `hello`, que l'on peut exécuter, ce qui donne :

```
you@machine$ ./hello
Hello the World !
```

Ici, la commande `gcc` (sans le `-c`) fait l'édition des liens avec toute la bibliothèque système C, qui contient entre autres la fonction `printf`.

Si on avait utilisé des fonctions de la bibliothèque C de fonctions mathématiques, il aurait fallu rajouter sur la même ligne de commande `-lm` (la bibliothèque s'appelle `libm`). Pour la bibliothèque `libpthread`, on aurait écrit `-lpthread`.

```
#include <stdio.h>
#include <math.h>

int main( void )
{
  printf( "pi=%f\n", M_PI );
  return 0;
}

you@machine$ gcc -c pi.c
you@machine$ gcc pi.o -o pi -lm
you@machine$ ./pi
pi=3.141593
```

Par curiosité, si vous souhaitez voir la sortie du préprocesseur, la commande est `gcc -E`. Si vous souhaitez voir le code assembleur (langage machine avant assemblage), tapez `gcc -S`.

On verra ensuite comment utiliser les `Makefile` pour éviter de retaper ces lignes de commande.

1.5 Eléments de base du langage

Un code C est découpé en fonctions. Les instructions sont toujours écrites dans le corps des fonctions. A l'extérieur des fonctions (c'est-à-dire au niveau global du module), vous pouvez déclarer de nouveaux types, de nouvelles variables, ou indiquer l'existence d'autres fonctions. Tout *programme* C a un point d'entrée. C'est la fonction `main`. Lorsque vous exécuterez votre programme, le flot d'exécution commencera par rentrer dans cette fonction. Dans cette fonction, vous effectuerez des opérations, calculs, mais vous pouvez bien sûr appeler d'autres fonctions.

Les blocs d'instructions sont placés entre accolades { }. Les instructions sont terminées par un point virgule `;`.

Pour créer des objets complexes, il faut d'abord disposer de types simples. Le C fournit des types de données simples :

- caractère : `char`
- entiers : `char`, `short`, `int`, `long`, `long long`
- nombres à virgule flottante : `float`, `double`, `long double`
- le type *pointeur* d'un autre type, qui est toujours représentée par une adresse en mémoire (sur 32 ou 64 bit selon la taille du bus d'adresse).

C99 rajoute le type `complex`, pas toujours implémenté dans les compilateurs. C ne garantit pas leurs tailles et leurs précisions, mais ne donne que des bornes.

On peut aussi définir des type scalaires avec `enum`.

```
enum Boolean {False, True};
enum Couleur {Bleu, Blanc, Rouge};
enum Sens    {Gauche, Haut, Droite, Bas};
```

Malgré cette relative pauvreté en type de données, le C permet de construire des types complexes à l'aide des constructions suivantes :

- les tableaux : répétition de taille donnée de variables du même type, rangées consécutivement en mémoire.
- les structures (agrégats ou enregistrements) : regroupent des variables de type quelconque.
- les unions, qui rassemblent au même endroit mémoire des types possiblement distincts.
- les pointeurs : qui permettent de stocker les adresses des variables.
- l'instruction `typedef` qui permet de nommer un nouveau type.

Comme tout langage, on peut déclarer des variables d'un type souhaité. La variable est caractérisée par son nom (ou identifiant). Une variable créée à l'extérieur de toute fonction est dite "globale". Elle vivra pendant toute la durée d'exécution du programme. Elle est potentiellement atteignable partout dans le programme. Une variable créée dans une fonction a une portée limitée à la fonction. De plus la variable disparaitra lorsque le flot d'exécution quittera cette fonction (plus précisément, ce niveau d'appel de la fonction). Les variables non initialisées ont une valeur indéterminée.

```
char s[ ] = "Bonjour !"; // variable globale de type tableau de caractères

void f()
{
   int i = 5; // variable locale de type entier (32 bits souvent).
   float x = 3.5f; // variable locale de type virg. flot. (32 bits souvent).

// i et x sont détruits ici.
}
```

En un sens, le langage C s'occupe de l'allocation mémoire des variables et de leur désallocation. Les variables, comme les paramètres, sont alloués en fait sur la *pile d'exécution* du processus. Néanmoins, on verra qu'il faudra un autre mécanisme (l'allocation dynamique, ou sur le *tas*) pour obtenir des structures de taille arbitraire.

1.6 Expressions

Une expression en C est le résultat d'un ensemble d'opérations. Une expression est donc équivalent à une valeur. Les plus fréquentes sont les expressions arithmétiques (formées à partir des opérateurs classiques `+-*/%`), le groupage par parenthésage, ainsi que d'éventuels appels de fonctions retournant un entier.

Ceci est une expression entière.

```
3*(4+5)-1
```

Il y a de plus des opérateurs de manipulation bit à bit (et `&`, ou `|`, xor `^`, non $\sim$, décalage gauche `<<` et droit `>>`).

De façon similaire, nous avons les expressions en virgule flottante (opérateurs `+-*/`). Nous avons aussi les expressions booléennes (opérateurs `&& || !`). A noter que le type booléen n'existe pas vraiment en C. On utilise un entier qui, s'il est nul, indique faux, sinon indique vrai.

Les opérateurs de comparaison entre nombres (`< <= == >= > !=`) forment une expression booléenne à partir de la comparaison de nombres.

1.7 Affectations

On peut affecter le résultat d'une expression à une variable — plus généralement, ce que l'on appelle une *lvalue* modifiable. On utilise l'opérateur `=` pour l'affectation. Le type de l'expression doit correspondre au type de la variable (ou *lvalue*).

```
float x = 3.5f + 2.1f;
int y = 3 + 4;
int z = 2.5f; // déconseillé, mais marche aussi !
              // Le C fait des conversions automatiques.
```

On peut aussi utiliser les opérateurs `+=,-=,*=,/=,%=,<<=,>>=`.

1.8 Instructions conditionnelles

On utilise le mot-clé `if`, suivi de parenthèses pour la condition et d'un bloc qui s'exécute que si l'expression booléenne ou arithmétique s'évalue à vrai. Eventuellement, on peut placer un mot-clé `else` suivi du bloc d'instructions à faire dans le cas où l'expression s'évalue à faux.

```
double x = ...; // un nombre
...
double x_abs;
if ( x >= 0 )
  x_abs = x;
else
  x_abs = -x;
// x_abs contient abs(x)
```

L'opérateur ternaire `?:` permet d'être très compact lorsque l'on cherche juste à distinguer 2 cas dans le calcul d'une expression. On écrirait ainsi :

```
double x_abs = ( x >= 0 ) ? x : -x;
```

Le `switch` permet d'exécuter des instructions par cas. Il faut que la variable de décision soit entière, ou de type énuméré.

```
// retourne le nombre de jours d'un mois, hors année bissextile.
int mois = ...; // un numéro de mois
int nbjours;
switch( mois )
{
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    nbjours = 31;
    break;
  case 2:
    nbjours = 28;
    break;
  default:
    nbjours = 30;
}
```

1.9 Structures répétitives

Il s'agit maintenant de blocs de contrôle pour répéter des groupes d'instructions. Il y en a trois en C :

- `while (` *expr-booléenne* `)` *instruction(s)*
- `do` *instruction(s)* `while (` *expr-booléenne* `);`
- `for (` *instr-init* `;` *expr-booléenne* `;` *instr-term* `)` *instruction(s)*

1.9.1 La boucle tant que (while)

`while (` *expr-booléenne* `)` *instruction(s)*

Le bloc d'instruction est répété tant que l'expression booléenne vaut vrai. Dès lors que l'expression s'évalue à faux, le flot d'exécution passe à la suite.

On veut par exemple afficher la table de conversion entre degrés Fahreneit et Celsius.

```
#include <stdio.h>

int main( void )
{
  float fahr;    // contiendra les degres fahreneit
  float celsius; // contiendra les degres celsius
```

```
  float min = 0.0f;   // borne inf. pour la table
  float max = 300.0f; // borne sup. pour la table
  float delta = 20.0f; // incrément entre chaque valeur

  fahr = min;
  while ( fahr <= max )
    {
      celsius = ( 5.0f / 9.0f ) * ( fahr - 32.0f );
      printf( "%f °F = %f °C\n", fahr, celsius );
      fahr += delta;
    }
  return 0;
}
```

1.9.2 La boucle pour (for)

`for` (*instr-init* ; *expr-booléenne* ; *instr-term*) *instruction(s)*

Cette boucle comporte trois éléments placés entre parenthèses juste après le mot-clé `for` pour déterminer combien de fois l'instruction est exécutée. *instr-init* est toujours exécutée une et une seule fois en entrée de boucle. Ensuite si *expr-booléenne* est vrai, alors l'instruction (ou le bloc) est exécutée, puis l'instruction de terminaison *instr-term* est exécutée. Le cycle reprend sur un test de *expr-booléenne* et ainsi de suite. Dès que *expr-booléenne* s'évalue à faux, le flot d'exécution quitte complètement le bloc `for`.

```
// Affichage des entiers de 0 à 99
int i;
for ( i = 0; i < 100; ++i )
  printf( "%d\n", i );
```

La conversion degrés Fahreneit et Celsius avec la boucle `for`.

```
#include <stdio.h>

int main( void )
{
  float fahr;    // contiendra les degres fahreneit
  float celsius; // contiendra les degres celsius

  float min = 0.0f;   // borne inf. pour la table
  float max = 300.0f; // borne sup. pour la table
  float delta = 20.0f; // incrément entre chaque valeur

  for ( fahr = min; fahr <= max; fahr += delta )
    {
      celsius = ( 5.0f / 9.0f ) * ( fahr - 32.0f );
      printf( "%f °F = %f °C\n", fahr, celsius );
    }
  return 0;
}
```

1.10 Entrées-sorties simples

Quasiment tous les programmes C importent le module `stdio.h` pour disposer de fonctions d'affichage sur le terminal ou de saisie de caractères. Il se trouve que sous Unix tout processus (et donc programme) a une entrée standard (généralement le clavier qui tape dans la console), une sortie standard (la console), et une erreur standard (en général la console aussi).

Ces fonctions permettent d'accéder directement à ces flux d'entrée ou de sortie. Les fonctions les plus utilisées sont :

— entrée : `getchar`, `scanf`, `fscanf`, `sscanf`, ...

— sortie : `putchar`, `puts`, `printf`, `fprintf`, `sprintf`, ...

Par exemple, `getchar()` lit un caractère sur l'entrée standard qu'il retourne. Si l'entrée standard est finie (fermée), retourne le caractère `EOF`.

```
#include <stdio.h>

int main( void )
{
  int nbc = 0;
  while ( getchar() != EOF )
    ++nbc;
  printf( "nb car lus = %d\n", nbc );
  return 0;
}
```

1.11 Le constructeur tableau `[]`

Comme dans beaucoup de langages, on peut déclarer des tableaux de variables du même type `T`. Il y a autant de variables créées que le nombre spécifié entre `[N]`. Toutes ces variables sont contigües en mémoire. On y accède via l'opérateur `[]`, en précisant un indice entre 0 et `N-1`. Attention, le tableau tout entier (et ses variables) disparait à la fin du bloc de définition.

1.12 Le constructeur `struct`

Le mot-clé `struct` permet d'agréger des variables de types quelconques afin de créer des types aussi complexes que nécessaires. On parle souvent d'enregistrement (*record* en anglais).

`struct` *struct-name* { *list-decl-var* } ;

list-decl-var est une liste de variables, dont les noms forment les noms des champs accessibles dans la structure. Un rationnel s'écrirait ainsi :

```
struct SRationnel {
  int num; /* numerateur */
  int den; /* denominateur */
};

/* On en fait un type comme les autres. */
typedef struct SRationnel Rationnel;

/* On déclare des variables. */
Rationnel a = { 3, 5 }; /* 3/5 */
Rationnel b = { 8, 13 }; /* 8/13 */

/* On accede aux champs avec la notation "pointee" "." */
a.num = b.num;
a.den = b.den; /* a vaut 3/5 maintenant */
```

On ne peut afficher directement une structure : il faut le faire champ à champ. On peut faire les affectations de structures, à ce moment-là, les valeurs des champs sont recopiées membre à membre. Les types des champs sont quelconques, ce qui permet de construire des types aussi complexes que souhaités. On note enfin l'utilisation du mot-clé `typedef` qui, en créant un alias, permet de manipuler les structures comme des types de base. Cela évite ainsi de répéter `struct` tout le temps.

Un bon exemple pour montrer des structures de données emboitées est de faire une structure `Point`, puis une structure `Triangle`. On pourra ensuite écrire les fonctions `distance` entre 2 points, `perimetre` et `aire` d'un triangle. On peut voir ensuite comment faire un polygone, avec le problème du nombre quelconque de côtés.

1.13 Exercices

Quelques exercices en vrac.

— Tableaux

1. Ecrire une fonction qui met à une valeur donnée tous les éléments d'un tableau.
2. Ecrire une fonction qui retourne les éléments d'un tableau.
3. Ecrire une fonction `int argmax( double tab[], int t )` qui calcule la position du plus grand élément d'un tableau de double de taille `t`.
4. En déduire la procédure de tri par sélection.

— Structures

1. Proposer une structure `Point` pour représenter un point du plan.
2. Ecrire une fonction qui retourne un point à partir de deux coordonnées x et y.
3. Ecrire une fonction qui retourne un point à partir des coordonnées polaires r et θ.
4. Ecrire la fonction qui retourne la distance euclidienne entre deux points.
5. Ecrire la fonction qui translate un point. Attention, il faut des pointeurs.

2 Fonctions, passage de paramètres, pointeurs

Cette section explicite comment définir de nouvelles fonctions, décrit le système de passage de paramètres en C, puis introduit les pointeurs. Les pointeurs sont en effet très vite rapidement nécessaires, notamment pour passer des paramètres en (entrée/)sortie (passage dit par adresse).

2.1 Déclaration et définition des fonctions

Pour utiliser une fonction, il faut qu'elle ait été déclarée au préalable. Le code associé à la fonction n'est pas nécessairement connu.

```
// déclaration de f
void f( int i ); // indique l'existence d'une fonction f,
                 // prenant un entier en paramètre.

int main( void )
{
  f( 7 ); // 1er appel de f
  f( 3 ); // 2ème appel de f
}

// Définition de f
void f( int i )
{
  printf( "%d au carré = %d\n", i, i*i );
}
```

2.2 Fonctions récursives

La façon dont les fonctions s'appellent les unes les autres forment un graphe, appelé graphe d'appel des fonctions. Lorsque ce graphe contient une boucle partant d'une fonction, on dit que la fonction est récursive. La récursivité n'est pas vraiment beaucoup plus lente qu'une boucle équivalente, quoique toute fonction récursive peut être transformée en boucle avec une structure de données auxiliaire (une pile). Elle permet surtout des écritures très concises et, pour qui est habitué, très clair. La récursivité est le socle de la programmation fonctionnelle.

Exemple : factorielle récursive.

Combien de fois les fonctions ont-elles été appelées dans les exemples ci-dessous ?

Fonction mystere1 (p x q par q + (p-1)*q)

mystere1(5, 3)

Fonction mystere2 (p x q par 2 * p/2 *q si p pair, et q + p/2 * 2 * q sinon)

mystere2(5, 3)

2.3 Passage de paramètres

Dans une fonction, les paramètres formels reçoivent les valeurs des arguments d'appel au moment de l'appel. Ce sont des variables locales, dont l'existence est bornée par le bloc de définition de la fonction.

Le C ne connaît que le passage par valeur (types de base, struct, union, pointeurs). Ainsi l'exemple classique de l'échange de variable ne marche pas.

```
void echange( int i, int j )
{
  int t = i;
  i = j;
  j = t;
}

int main( void )
{
  int x = 4;
  int y = 8;
  echange( x, y );
  // x et y sont inchangés.
}
```

2.4 Passage par adresse

Curieusement, lorsqu'on change les valeurs d'un tableau, ces changements sont bien répercutés. Cela provient du fait que le C fait bien le passage par valeur du tableau, mais comme le tableau est défini comme étant l'adresse de la première case, seule l'adresse en mémoire du tableau est passée par valeur.

```
void echange( char tab[], int i, int j )
{
  int t = tab[i];
  tab[i] = tab[j];
  tab[j] = t;
}

int main( void )
{
  char tab[] = "Bonjour";
  int x = 2;
  int y = 6;
  echange( tab, x, y );
  // tab contient "Borjoun"
}
```

Pour passer des paramètres en E/S, on fera donc la même chose. On passera en paramètre l'adresse de la variable à modifier. Cela se fait via les opérateurs '&' et '*'.

```
void echange( int* i, int* j )
{
  int t = *i;
  *i = *j;
  *j = t;
}

int main( void )
{
  int x = 4;
  int y = 8;
```

```
  echange( &x, &y );
  // x et y sont changés: x=8, y=4.
}
```

2.5 Les pointeurs

On appelle *pointeur* un couple adresse en mémoire/type de l'élément pointé. Une variable dont le type permet de stocker un pointeur est appelée *variable pointeur*. C'est une variable comme les autres, sauf que le nombre qu'elle contient désigne une adresse en mémoire. Sa taille en octet dépend donc de la taille du bus d'adresse de votre carte mère.

L'*opérateur d'adresse '&'* appliqué à une variable (en fait à toute lvalue, c'est-à-dire un identifiant désignant une case en mémoire) retourne un pointeur qui est l'adresse de la variable, et dont le type est le type de la variable.

```
#include <stdio.h>

int gt[ 10 ];

int main( void )
{
  int t[ 10 ];
  double x;
  double y;
  printf( "t = %p\n", t );
  printf( "&t = %p\n", &t );
  printf( "&t[ 0 ] = %p\n", &t[ 0 ] );
  printf( "&t[ 9 ] = %p\n", &t[ 9 ] );
  printf( "gt = %p\n", gt );
  printf( "&gt = %p\n", &gt );
  printf( "&gt[ 0 ] = %p\n", &gt[ 0 ] );
  printf( "&gt[ 9 ] = %p\n", &gt[ 9 ] );
  printf( "&x = %p\n", &x );
  printf( "&y = %p\n", &y );
  return 0;
}
```

Le programme précédent affiche le résultat suivant

```
t = 0x7fff4f591880
&t = 0x7fff4f591880
&t[ 0 ] = 0x7fff4f591880
&t[ 9 ] = 0x7fff4f5918a4
gt = 0x601040
&gt = 0x601040
&gt[ 0 ] = 0x601040
&gt[ 9 ] = 0x601064
&x = 0x7fff4f5918b8
&y = 0x7fff4f5918b0
```

On voit déjà que les variables globales ne sont pas allouées au même endroit que les variables locales.

Pour déclarer une variable pointeur pouvant contenir un pointeur pointant sur un type `t`. On place *l'étoile* * après le type.

```
int i = 17;
int* ptr = &i;   // valide
if ( i == *ptr ) // vrai, car la valeur pointée par ptr correspond à la case i
 ...
```

Pour accéder à la valeur, on utilise l'*opérateur d'indirection '*'*. Utilisé à droite dans une expression, cet opérateur retourne la valeur pointée. Utilisé à gauche dans une affectation, cet opérateur correspond à désigner la variable à cette adresse. Cela imite parfaitement le rôle des variables, qui ne sont que des valeurs à droite, et qui sont des variables à gauche dans une affectation.

Bien sûr, un pointeur vers T ou (un T*) est un type. On a donc le droit de faire des pointeurs vers pointeurs etc.

Exemple des arguments de la ligne de commande.

2.6 Les tableaux sont (presque toujours) des pointeurs

En fait, une variable de type tableau est essentiellement un pointeur vers sa première case. Il a juste la propriété que l'on ne peut pas changer sa valeur. (e.g. ++t n'a pas de sens, alors qu'il a un sens pour un pointeur).

On note que l'opérateur & sur un tableau t retourne la même chose que t tout court. Il est donc inutile de passer son adresse pour un passage par adresse.

Les autres différences résident surtout dans l'initialisation. On note par exemple le cas spécifique des tableaux de caractères :

```
char* nom1 = "Borjour";  /* alloué dans la section DATA, nom1 pointe donc
                            vers une zone non modifiable. */
nom1[ 2 ] = 'n';         /* est donc invalide;
char nom2[] = "Borjour"; /* alloué sur la pile, nom2 pointe donc
                            vers une zone modifiable. */
nom2[ 2 ] = 'n';         /* est donc valide;
```

Une remarque importante est que l'opérateur d'accès à un indice `[]` et l'opérateur d'indirection `*` sont très similaires. En fait, on a, si `t` désigne un tableau (ou un pointeur) :

`t[ i ]` $\Leftrightarrow$ `*(t + i)`

En particulier, l'accès au premier élément d'un tableau est tout simplement l'indirection : `t[ 0 ]` $\Leftrightarrow$ `*t`

Exemple : copier un tableau dans un autre. Montrer qu'on peut appeler sur un même tableau, par exemple pour décaler les données.

2.7 Arithmétique des pointeurs

Il est légal d'appliquer des additions ou soustractions sur des pointeurs. L'adresse est décalée d'autant de cases que l'entier indiqué. On peut aussi comparer les pointeurs avec les opérateurs de comparaison usuels. Attention, cela n'a de sens que si les pointeurs pointent vers des zones mémoire comparables, par exemple au sein d'un tableau ou d'un bloc mémoire alloué.

Les opérateurs suivant sont valides, si `p,q` sont des pointeurs et `i` un entier : `p++`,`++p`, `p--`, `--p`, `p!=q`, `p==q`, `p+i`, `p-i` entier.

On utilise l'opérateur `sizeof` pour savoir quelle mémoire utilise réellement un type.

2.8 Le pointeur `void*`

Il désigne un pointeur vers n'importe quoi. On peut le convertir en n'importe quel autre pointeur, à charge pour l'utilisateur de ne pas faire n'importe quoi.

```
#include <stdio.h>

int main( void )
{
  int i = 19;
  float x = 16.7;
  void* ptr = &i;
  int* ptr_int = (int*) ptr;
  ptr = &x;
  float* ptr_float = (float*) ptr;
```

```
  printf( "(*ptr_int) = %d\n", *ptr_int );
  printf( "(*ptr_float) = %f\n", *ptr_float );
  printf( "(*(int*)) = %x\n", * ((int*) ptr) );
  return 0;
}

(*ptr_int) = 19
(*ptr_float) = 16.700001
(*(int*)) = 4185999a
```

On peut voir ainsi le codage binaire des nombres flottants.

Exemple d'un copieur rapide :

```
typedef unsigned char octet;
typedef long long int bloc;

void fast_copy( void* dst, void* src, int n )
{
  int b = n / sizeof( bloc );
  bloc* bsrc = src;
  bloc* bdst = dst;
  while ( b-- )
    *bdst++ = *bsrc++;
  int o = n % sizeof( bloc );
  octet* osrc = bsrc;
  octet* odst = bdst;
  while ( o-- )
    *odst++ = *osrc++;
}

typedef struct {
  char nom[ 100 ];
  char prenom[ 100 ];
  int age;
  double taille;
} Personne;

int main( void )
{
  Personne P1, P2;
  ...
  fast_copy( &P2, &P1, sizeof( Personne ) );
}
```

2.9 Allocation dynamique

Comment manipuler des objets de taille variable ? Par exemple, vous voulez fabriquer une nouvelle chaîne de caractères qui est l'union de deux chaînes. Malheureusement, vous ne connaissez pas la taille des chaînes respectives. Autre problème, vous voulez renvoyer le contenu de variables locales, par exemple un tableau entier. La valeur de retour ne marcherait pas ici.

```
#include <stdio.h>

int* f()
{
  int t[ 10 ];
  return t; // en fait, à proscrire
}
```

```
void affiche( int* t )
{
  int i;
  for ( i = 0; i < 10; ++i )
    printf( "%d ", t[ i ] );
  printf("\n");
}

int main( void )
{
  int* tab = f();
  int i;
  for ( i = 0; i < 10; ++i )
    tab[ i ] = 0;
  affiche( tab );
  return 0;
}
```

Ce programme affiche (!) ce résultat surprenant.

```
0 0 4195766 0 0 0 821579552 32767 0 0
```

On utilise alors des fonctions spécifiques pour allouer ou désallouer de la mémoire. Ces blocs mémoire sont alloués au processus, et seront détruits automatiquement par le système à la fin de l'exécution du programme. Néanmoins, il vaut mieux se charger soi-même de la désallocation des blocs mémoire lorsqu'on n'en a plus besoin.

```
#include <stdlib.h>

void* calloc(size_t nmemb, size_t size);
void* malloc(size_t size);
void free(void* ptr);
void* realloc(void* ptr, size_t size);
```

Traditionnellement, on se sert de `malloc` et `free`.

- `malloc` : alloue un bloc mémoire sur le tas et retourne son adresse.
- `free` : désalloue le bloc mémoire à l'adresse spécifiée (sur le tas).
- `calloc` : alloue un tableau de `nmemb` éléments de taille spécifiée et l'initialise à 0.
- `realloc` : réalloue un bloc mémoire (en général souhaité plus grand).

Le tas (*heap* en anglais) est une autre zone mémoire possible pour le processus. Contrairement à la *pile d'exécution*, sa gestion est indépendante de l'entrée ou de la sortie d'une fonction et reste toujours spécifiée par les fonctions ci-dessus.

```
// ptr3.c
// correction de l'exemple précédent avec malloc/free.
#include <stdlib.h>
#include <stdio.h>

int* f()
{
  int* t = (int*) malloc( 10 * sizeof( int ) );
  return t; // ok
}

void affiche( int* t )
{
  int i;
  for ( i = 0; i < 10; ++i )
    printf( "%d ", t[ i ] );
  printf("\n");
}
```

```
int main( void )
{
  int* tab = f();
  int i;
  for ( i = 0; i < 10; ++i )
    tab[ i ] = 0;
  affiche( tab );
  // (*)
  return 0;
}
```

Affichera bien

```
0 0 0 0 0 0 0 0 0 0
```

En revanche, comme on a oublié de libérer l'espace mémoire, cela crée une fuite mémoire (memory leak). Un outil comme valgrind le détecte ainsi :

```
[you] gcc ptr3.c -o ptr3
[you] valgrind ./ptr3
==27132== Memcheck, a memory error detector
==27132== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==27132== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h for copyright info
==27132== Command: ./ptr3
==27132==
0 0 0 0 0 0 0 0 0 0
==27132==
==27132== HEAP SUMMARY:
==27132==     in use at exit: 40 bytes in 1 blocks
==27132==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==27132==
==27132== LEAK SUMMARY:
==27132==    definitely lost: 40 bytes in 1 blocks
==27132==    indirectly lost: 0 bytes in 0 blocks
==27132==      possibly lost: 0 bytes in 0 blocks
==27132==    still reachable: 0 bytes in 0 blocks
==27132==         suppressed: 0 bytes in 0 blocks
==27132== Rerun with --leak-check=full to see details of leaked memory
==27132==
==27132== For counts of detected and suppressed errors, rerun with: -v
==27132== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

On peut donc le corriger en rajoutant la ligne (au niveau de (*))

```
free( tab );
```

Ce qui donnerait alors :

```
[you] gcc ptr3.c -o ptr3
[you] valgrind ./ptr3
==27260== Memcheck, a memory error detector
==27260== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==27260== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h for copyright info
==27260== Command: ./ptr3
==27260==
0 0 0 0 0 0 0 0 0 0
==27260==
==27260== HEAP SUMMARY:
```

```
==27260==     in use at exit: 0 bytes in 0 blocks
==27260==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==27260==
==27260== All heap blocks were freed -- no leaks are possible
==27260==
==27260== For counts of detected and suppressed errors, rerun with: -v
==27260== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

Comme on le voit, on peut allouer des blocs mémoire de n'importe quelle taille. On peut donc définir des tableaux de taille choisie à l'exécution.

```
// Fonction similaire à strdup.
char* duplicate( const char * str )
{
  int n = 0;
  while ( str[ n ] != '\0' ) ++n;
  // le n+1 est pour ajouter le '\0' à la fin.
  char* str2 = (char*) malloc( (n+1) * sizeof( str ) );
  int i;
  for ( i = 0; i <= n; ++i )
    str2[ i ] = str[ i ];
  return str2;
}

int main( void )
{
  char s[] = "Bonjour Toto";
  char* s2 = duplicate( s );
  s[10] = 'i';
  s[12] = 'i';
  printf( "%s, %s\n", s, s2 ); /* Bonjour Toto, Bonjour Titi */
  free( s2 );
}
```

2.10 La pile d'exécution du processus

On peut maintenant regarder en détails comment le C exécute les fonctions et alloue les variables locales. On comprendra mieux pourquoi les variables locales ont cette portée. On reprend l'exemple de `duplicate`.

2.11 Pointeurs sur les structures

On peut bien sûr manipuler des pointeurs vers une structure. Si la structure `T` contient un champ `name`, alors si `p` est un pointeur de `T`, `(*p).name` accède au champ `name`. On dispose aussi de la notation raccourcie `p->name`, dite notation flèche, à préférer en pratique car plus lisible.

Un bon exercice ici est sans doute une struct `String` complète, bien sécurisée. Nécessite de l'allocation dynamique, des pointeurs `->`, on peut sécuriser les opérateurs...

2.12 Un exemple complet : les piles en C

Faire l'exemple complet des piles, avec agrandissement si la capacité de la pile est atteinte.

Cela permet d'expliciter l'usage des pointeurs sur les structures, du passage par adresse, de l'allocation dynamique, ainsi que les bonnes pratiques pour manipuler les structures en C.

On pourra appliquer la pile au calcul de la suite de Fibonacci.

```
#include <stdio.h>

struct SPile {
  int elems[ 100 ]; // stockera les éléments
```

```
  int nb; // nb elements
};
typedef struct SPile Pile;

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

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

int Pile_pleine( Pile* p )
{
  return p->nb == 100;
}

void Pile_push( Pile* p, int e )
{
  if ( ! Pile_pleine( p ) )
    {
      p->elems[ p->nb++ ] = e;
    }
  else
    printf( "Pile_push( Pile* p, int e ): Pile pleine.\n" );
}

int Pile_sommet( Pile* p )
{
  if ( ! Pile_vide( p ) )
    {
      return p->elems[ p->nb - 1 ];
    }
  else
    printf( "Pile_sommet( Pile* p ): Pile vide.\n" );
}

void Pile_pop( Pile* p )
{
  if ( ! Pile_vide( p ) )
    {
      --p->nb;
    }
  else
    printf( "Pile_pop( Pile* p ): Pile vide.\n" );
}

// 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 adaptera la structure précédente pour obtenir une taille quelconque, grâce à l'allocation dynamique.

2.13 Tableaux comme si ce n'étaient pas des pointeurs

Si vous voulez manipuler les tableaux comme les autres variables (passage par valeur, affectation), une possibilité est simplement de les encapsuler dans une structure.

```
typedef struct STabInt100 {
  int elems[ 100 ];
} TabInt100;

...
TabInt100 t1;
TabInt100 t2;
...
t1 = t2; // valide : copie tous les éléments du tableau.
...
TabInt100 f( ... ) ...
...
t1 = f( ... ); // valide : la valeur de retour est un gros tableau.
...
g( TabInt100 t ) // valide, mais coûteux : le tableau sera recopié.
{ ... }
```

Soyez donc vigilant dans les passages de paramètres, de façon à ne copier des grosses données que lorsque c'est nécessaire.

2.14 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).

```
#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
```

2.15 Structures auto-référentes

Par exemple, la liste chaînée ou l'arbre binaire. *Work in progress.*

2.16 Exercices

1. On vous donne le code suivant :

```
#include <stdio.h>

void echange( char tab[], int i, int j )
{
  int t = tab[i];
  tab[i] = tab[j];
  tab[j] = t;
}

typedef struct SString {
  char tab[10];
} String;

void echange2( String s, int i, int j )
{
  int t = s.tab[i];
  s.tab[i] = s.tab[j];
  s.tab[j] = t;
}

int main( void )
{
  char tab[10] = "Bonjour\0";
  int x = 2;
  int y = 6;
  echange( tab, x, y );
  // tab contient "Borjoun"
```

```
  printf( "%s\n", tab );
  String s = { "Bonjour\0" };
  echange2( s, x, y );
  // s.tab contient "Bonjour"
  printf( "%s\n", s.tab );
  return 0;
}
```

Lorsqu'on l'exécute, on obtient :

```
Borjoun
Bonjour
```

Expliquez la différence dans le résultat. Pourquoi dans un cas la chaîne est modifiée alors qu'elle reste inchangé dans le second. Proposez quelques légères modifications pour que le programme retourne deux fois `borjoun`.

2. On veut faire une structure "chaîne de caractères" en C. Proposez un tel type de données, basé sur un pointeur vers caractère, un entier donnant la taille de la chaîne, un entier donnant la taille réellement allouée. Ecrivez le type `Chaine`, les fonctions `Chaine_init`, `Chaine_concat`, `Chaine_taille`, `Chaine_toC`, `Chaine_at`, etc.

2.17 Les pointeurs de fonction

Le nom d'une fonction est assimilable à l'adresse de son code source. On peut donc considérer un nom de fonction comme un pointeur, dont le type est ainsi formé :

```
TypeRetour NomFonction( Type1 param1, Type2 param2 );
typedef TypeRetour (*TypeFonction)( Type1 param1, Type2 param2 );
```

`TypeFonction` désigne alors un pointeur vers une fonction qui a deux paramètres en entrée (de types `Param1` et `Param2`), et qui retourne un `TypeRetour`. On peut par exemple faire ceci :

```
double addition( double x, double y )
{
  return x+y;
}
double multiplication( double x, double y )
{
  return x*y;
}
typedef double (*OperationBinaire)( double x, double y ); /* x,y facultatifs */

int main( void )
{
  double a = 10.0;
  double b = 0.5;
  OperationBinaire op = addition; /* &addition idem */
  printf( "%g\n", op(a,b) ); /* affiche 10.5 */
  op = multiplication; /* &multiplication idem */
  printf( "%g\n", op(a,b) ); /* affiche 5.0 */
  return 0;
}
```

Les pointeurs de fonction forment en C la brique de base pour modifier le comportement des structures de données, pour avoir des réactions à des événements/signaux personnalisés, plus généralement pour implémenter le polymorphisme. On verra leur utilisation dans GTK pour définir des fonctions appelées lorsque certains événements se produisent (clic souris, time-out, etc).

Quelques exemples d'utilisation

— Appliquer une fonction à une collection d'éléments

— Définir une fonction *callback* pour une structure. C'est très utilisé en IHM, où on associe aux élément graphiques (fenêtre, zone de dessin, etc) des fonctions qui sont appelées pour un événement précis (clic souris, déplacement de la souris, ordre de réaffichage).

```
#include <stdlib.h>
#include <stdio.h>

typedef void (* FctOnDouble)( double* );
void reset( double* x )
{ *x = 0.0; }
void uniform( double* x )
{ *x = (double) rand() / (double) RAND_MAX; }
void add1( double* x )
{ *x += 1; }
void square( double* x )
{ *x *= *x; }
void affiche( double*x )
{ printf(" %f", *x ); }
void apply( double* begin, double* end, FctOnDouble f )
{
  for ( ; begin != end; ++begin )
    f( begin );
}
int main()
{
  double t[ 5 ];
  apply( t, t+5, reset );
  apply( t, t+3, uniform );
  apply( t+2, t+5, add1 );
  apply( t+4, t+5, add1 );
  apply( t, t+5, square );
  apply( t, t+5, affiche );
  printf("\n");
  return 0;
}
```

Comment faire un accumulateur, un moyenneur ? Que faut-il rajouter en plus ?

3 Usages et pratiques de la programmation en C

3.1 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

```
#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

```
#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

```
#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
```

3.2 Le préprocesseur

Avant d'être compilé, le fichier source est transformé en un autre fichier texte (plus long en général) par ce qu'on appelle le préprocesseur. Celui-ci va détecter et interpréter les commandes commençant par '#'. Ensuite, ces commandes/définitions influent sur la sortie du préprocesseur. Les commandes les plus importantes sont :

`#include` ***file-name*** Dis au préprocesseur de copier/coller le fichier *file-name* dans ce source à cet endroit-là.

`#define` ***name value*** Dis au préprocesseur de définir une variable de substitution. Les variables de substitution du préprocesseur fournissent un moyen simple de nommer des constantes. En effet :

```
#define CONSTANTE valeur
```

permet de substituer presque partout dans le code source qui suit cette ligne la suite de caractères "CONSTANTE" par la valeur. Plus précisément, la substitution se fait partout, à l'exception des caractères et des chaines de caractères. Par exemple, dans le code suivant :

```
#define TAILLE 100
printf("La constante TAILLE vaut %d\n", TAILLE);
```

La substitution se fera sur la deuxième occurrence de TAILLE, mais pas la première. Le préprocesseur transformera ainsi l'appel à printf :

```
printf("La constante TAILLE vaut %d\n", 100);
```

Le préprocesseur procède à des traitements sur le code source, sans avoir de connaissance sur la structure de votre programme. Dans le cas des variables de substitution, il ne sait faire qu'un remplacement de texte, comme le ferait un traitement de texte. On peut ainsi les utiliser pour n'importe quoi, des constantes, des expressions, voire du code plus complexe.

`#define` ***macro-name*(...) *expr*** Une macro est en fait une constante qui peut prendre un certain nombre d'arguments. Les arguments sont placés entre parenthèses après le nom de la macro sans espaces, par exemple :

```
#define MAX(x,y) x > y ? x : y
#define SWAP(x,y) x ^= y, y ^= x, x ^= y
```

La première macro prend deux arguments et "retourne" le maximum entre les deux. La deuxième est plus subtile, elle échange la valeur des deux arguments (qui doivent être des variables entières), sans passer par une variable temporaire, et ce avec le même nombre d'opérations.

`#if ... #else ... #endif` Permettent la compilation conditionnelle. Ainsi

```
#if defined( DEBUG )
  assert( x > 0 );
#endif
```

Permet de ne vérifier la valeur de x que dans le mode `DEBUG`. On peut faire plus concis et plus pratique sous la forme suivante :

```
#if defined( DEBUG )
#define ASSERT( e ) assert( e )
#else
#define ASSERT( e ) /* vide ! */
#endif
...
  ASSERT( x > 0 ); /* ASSERT sera substitué automatiquement par
                      une instruction vide ou la fonction assert
                      selon que DEBUG est défini ou non. */
```

La commande `#ifdef` *name* est équivalente à `#if defined(` *name* `)`

On se sert beaucoup de ces commandes pour éviter qu'un fichier en-tête ne soit inclus plusieurs fois lors de la compilation d'un fichier source.

`#pragma` Appelle des commandes spécifiques, pas forcément partagées par tous les compilateurs. Cela permet parfois de supprimer des "warning"s de compilation que vous savez sans conséquences. On utilise beaucoup le pragma suivant en début de fichier entête pour éviter de l'inclure plusieurs fois.

```
#pragma once
```

3.3 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.

3.4 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 :

```
/* 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
```

3.5 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.  */
voir 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).

3.6 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.

3.6.1 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 :

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

```
} Personne;

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

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.

3.6.2 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 :

```
#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 doute préferrable 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).

3.7 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).

```
/* 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;
}
```

3.8 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 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`.

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_*/
```

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.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
```

3.9 Autres spécificités du C

— les `union` permettent de définir des types à choix.
— on peut définir des fonctions avec nombre variable d'arguments, i.e. *variadic functions* (header `stdarg.h`, notation `...`, type `va_list`, commandes `va_start`, `va_arg`, `va_end`). La fonction `printf` est certainement l'exemple le plus utilisé.
— pas de surcharge : un nom de fonction = une fonction. Le seul moyen de simuler plusieurs fonctions qui ont le même nom est d'utiliser les fonctions avec nombre variable d'arguments.