Université Savoie Mont Blanc,
UFR SceM
INFO504, L3 INFO

September 29, 2020
J.-O. Lachaud

Fiche 0 — Rappels rapides de langage C

(Historique, principes, compilation, types, expressions, instructions conditionnelles et répétitives)

*Cette fiche est à lire **avant** le premier cours.*

1 Préambule

Cette fiche donne quelques rappels brefs sur le langage C, puisque vous avez déjà suivi un cours de C en L2, ainsi que des cours de programmation impérative. Il vous appartient de vérifier que vous avez bien acquis les principaux points rappelés dans ce document. Dans les fiches suivantes, nous aborderons plus en détails certains éléments clés (et parfois délicats) du C, comme les tableaux, les pointeurs, l'allocation dynamique, les structures, les pointeurs de fonction.

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.

Ce document ne donnera pas tous les éléments de la syntaxe du C, loin de là. Il existe des ressources disponibles qui le font. Par exemple, *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`.

2 Introduction au langage C

2.1 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, Rust 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).

2.2 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).

2.3 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.

- Malgré presque 50 ans d'existence, le langage C est encore très utilisé, et surtout présent sur tous les systèmes.
- Sa connaissance est indispensable pour la programmation bas niveau (systèmes embarqués, systèmes d'exploitation, pilotes de périphériques, interpréteurs de langages).

- Le C a eu une influence profonde sur de nombreux langages de programmation, même assez haut niveau.
- Le C est un langage proche de l'assembleur tout en restant portable et générique. Comprendre le C c'est comprendre comment un microprocesseur exécute un programme.
- Une bonne connaissance de C est indispensable pour bien comprendre le C++.
- Le principal défaut du C est son passage à l'échelle, même s'il existe de gros logiciels et bibliothèques écrits entièrement en C. Au final, le C demande plus de discipline que des langages de plus haut niveau.

3 Cycle de développement d'un programme C

Le code C est écrit dans deux sortes de fichiers texte:

- les fichiers *sources*, terminés par le suffixe `.c`, qui contiennent le code des fonctions, la définition des variables globales, et éventuellement des déclarations de type propre à ce fichier.
- les fichiers *entêtes*, terminés par le suffixe `.h`, qui contiennent les déclarations des fonctions et des types, ainsi que les déclarations des variables globales que l'on souhaite rendre public.

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`:

```
// hello.c
#include <stdio.h>

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

On écrira pour compiler ce fichier sur votre shell (en fait 2 étapes d'un coup: préprocesseur, compilation) :

```
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
```

L'éditions des liens requiert *une* fonction `main`, définie dans un des fichiers objet. 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.

- On écrit du C dans deux types de fichiers texte: les entêtes (.h) pour les déclarations (ça existe), les sources pour les définitions et le code (voilà le code associé).
- préprocesseur: `gcc -E source.c`
- préprocesseur + compilation: `gcc -c source1.c source2.c ...`
- éditions des liens: `gcc source1.o source2.o ... -o prog`
- préprocesseur + compilation + éditions des liens: `gcc source1.c source2.c ... -o prog`
- ajout de bibliothèques standards (e.g. `libm.a` dans `/usr/lib`): `gcc ... -lm`
- ajout de bibliothèques extérieures (e.g. `libjpeg.a` dans `/usr/local/lib`):: `gcc ... -L/usr/local/lib -ljpeg`

4 Eléments de base du langage

4.1 Structuration en fonctions et blocs d'instructions

Un code C est découpé en *fonctions*. Les instructions sont toujours écrites dans le corps des fonctions, c'est-à-dire le bloc d'instructions qui suit la déclaration de la fonction. 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 `;`.

4.2 Types et variables

En C, les données sont typées et sont placées dans des variables typées. 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 int`, `long long int`
- 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.

4.3 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.

4.4 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 `+=,-=,*=,/=,%=,<<=,>>=`.

4.5 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;
}
```

4.6 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)*

4.6.1 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;
}
```

4.6.2 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;
}
```

4.7 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 pro-

gramme) 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`.

```c
#include <stdio.h>

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

Les fonctions `printf` et `scanf` et leurs variantes sont bien utiles. Leur fonctionnement suit un principe commun. Une chaîne de caractères décrit les paramètres suivants qui doivent être affichés ou lus.

Pour `printf`, la chaîne donnent aussi le texte à afficher. Les paramètres à afficher sont indiqués par `%` suivi de lettre(s) et chiffre(s) pour préciser le paramètre. Par exemple:

- `%d` indique un paramètre de type entier (`int`)
- `%f` indique un paramètre de type nombre à virgule flottant (`float`)
- `%c` indique un paramètre de type caractère (`char`)
- `%s` indique un paramètre de type chaîne de caractère (`char*`), terminée par le caractère 0.
- `%p` indique un paramètre de type pointeur (`void*`, `int*`, ...)
- `%ld` indique un paramètre de type entier long (`long int`)
- `%lf` indique un paramètre de type nombre à virgule flottant double précision (`double`)
- `%03d` indique un paramètre de type entier, affiché avec au moins 3 chiffres et complété devant par des 0.
- etc.

```c
int     a = 5;
double pi = 3.14;
char*   s = "Toto";

// Affiche "M Toto possède 5 cordes à son arc"
printf( "M %s possède %d cordes à son arc\n", s, a );

// Affiche "La valeur de pi est 3.14"
printf( "La valeur de pi est %d\n", pi );

// Ecrit "La valeur de pi est 3.14" dans le tableau buffer.
char buffer[ 100 ];
sprintf( buffer, "La valeur de pi est %d\n", pi );
```

La fonction `scanf` permet de lire des valeurs au clavier et de les placer dans des variables, sa variante `sscanf` permet de lire des valeurs dans une chaîne de caractères. Ses fonctions retournent le nombre de champs bien lus.

```c
int a, b, c;
char s[] = "17 54 32";
int    n = sscanf( s, "%d %d %d", &a, &b, &c );
// a=17, b=54, c=32, n=3
```