Le langage C

Différents langages permettent au programmeur de construire des programmes qui seront exécutés par le processeur. En réalité, le processeur ne comprend qu'un langage : le langage machine. Ce langage est un langage binaire dans lequel toutes les commandes et toutes les données sont représentés sous la forme de séquences de bits.

Le langage machine est peu adapté aux humains et il est extrêmement rare qu'un informaticien doive manipuler des programmes directement en langage machine. Par contre, pour certaines tâches bien spécifiques, comme par exemple le développement de routines spéciales qui doivent être les plus rapides possibles ou qui doivent interagir directement avec le matériel, il est important de pouvoir efficacement générer du langage machine. Cela peut se faire en utilisant un langage d'assemblage. Chaque famille de processeurs a un langage d'assemblage qui lui est propre. Le langage d'assemblage permet d'exprimer de façon symbolique les différentes instructions qu'un processeur doit exécuter. Nous aurons l'occasion de traiter à plusieurs reprises des exemples en langage d'assemblage dans le cadre de ce cours. Cela nous permettra de mieux comprendre la façon dont le processeur fonctionne et exécute les programmes. Le langage d'assemblage est converti en langage machine grâce à un **assembleur**.

Le langage d'assemblage est le plus proche du processeur. Il permet d'écrire des programmes compacts et efficaces. C'est aussi souvent la seule façon d'utiliser des instructions spéciales du processeur qui permettent d'interagir directement avec le matériel pour par exemple commander les dispositifs d'entrée/sortie. C'est essentiellement dans les systèmes embarqués qui disposent de peu de mémoire et pour quelques fonctions spécifiques des systèmes d'exploitation que le langage d'assemblage est utilisé de nos jours. La plupart des programmes applicatifs et la grande majorité des systèmes d'exploitation sont écrits dans des langages de plus haut niveau.

Le langage **C [KernighanRitchie1998]**, développé dans les années 70 pour écrire les premières versions du système d'exploitation **Unix**, est aujourd'hui l'un des langages de programmation les plus utilisés pour développer des programmes qui doivent être rapides ou doivent interagir avec le matériel. La plupart des systèmes d'exploitation sont écrits en langage C.

Le langage C a été conçu à l'origine comme un langage proche du processeur qui peut être facilement compilé, c'est-à-dire traduit en langage machine, tout en conservant de bonnes performances.

La plupart des livres qui abordent la programmation en langage C commencent par présenter un programme très simple qui affiche à l'écran le message *Hello, world!*.

```c
/************************************
 * Hello.c
 *
 * Programme affichant sur la sortie
 * standard le message "Hello, world!"
 *
 ************************************/

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

int main(int argc, char *argv[])
{
   // affiche sur la sortie standard
   printf("Hello, world!\n");

   return EXIT_SUCCESS;
}
```

Pour être exécuté, ce programme doit être compilé. Il existe de nombreux compilateurs permettant de transformer le langage C en langage machine. Dans le cadre de ce cours, nous utiliserons **gcc(1)**. Dans certains cas, nous pourrons être amenés à utiliser d'autres compilateurs comme **llvm**.

La compilation du programme **`src/hello.c`** peut s'effectuer comme suit sur une machine de type Unix :

```
$ gcc -Wall -o hello hello.c
$ ls -l
total 80
-rwxr-xr-x  1 obo  obo  8704 15 jan 22:32 hello
-rw-r--r--  1 obo  obo   288 15 jan 22:32 hello.c
```

gcc(1) supporte de très nombreuses options et nous aurons l'occasion de discuter de plusieurs d'entre elles dans le cadre de ce cours. Pour cette première utilisation, nous avons choisi l'option *-Wall* qui force **gcc(1)** à afficher tous les messages de type **warning** (dans cet exemple il n'y en a pas) et l'option *-o* suivie du nom de fichier *hello* qui indique le nom du fichier dans lequel le programme exécutable doit être sauvegardé par le compilateur **[1]**.

Lorsqu'il est exécuté, le programme *hello* affiche simplement le message suivant sur la sortie standard :

```
$ ./hello
Hello, world!
$
```

Même si ce programme est très simple, il illustre quelques concepts de base en langage C. Tout d'abord comme en Java, les compilateurs récents supportent deux façons d'indiquer des commentaires en C :

- un commentaire sur une ligne est précédé des caractères //
- un commentaire qui comprend plusieurs lignes débute par /* et se termine par */

Ensuite, un programme écrit en langage C comprend principalement des expressions en langage C mais également des expressions qui doivent être traduites par le **préprocesseur**. Lors de la compilation d'un fichier en langage C, le compilateur commence toujours par exécuter le préprocesseur. Celui-ci implémente différentes formes de macros qui permettent notamment d'inclure des fichiers (directives `#include`), de compiler de façon conditionnelle certaines lignes ou de définir des constantes. Nous verrons différentes utilisations du préprocesseur C dans le cadre de ce cours. À ce stade, les trois principales fonctions du préprocesseur sont :

- définir des substitutions via la macro `#define`. Cette macro est très fréquemment utilisée pour définir des constantes ou des substitutions qui sont valables dans l'ensemble du programme.

```
#define ZERO  0
#define STRING "LEPL1503"
```

- importer (directive `#include`) un fichier. Ce fichier contient généralement des prototypes de fonctions et des constantes. En langage C, ces fichiers qui sont inclus dans un programme sont appelés des *header files* et ont par convention un nom se terminant par `.h`. Le programme **`src/hello.c`** ci-dessus importe deux fichiers *headers* standards :
  - `<stdio.h>` : contient la définition des principales fonctions de la librairie standard permettant l'interaction avec l'entrée et la sortie standard, et notamment **printf(3)**
  - `<stdlib.h>`: contient la définition de différentes fonctions et constantes de la librairie standard et notamment `EXIT_SUCCESS` et `EXIT_FAILURE`. Ces constantes sont définies en utilisant la macro `#define` du préprocesseur

```
#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0
```

- inclure du code sur base de la valeur d'une constante définie par un `#define`. Ce contrôle de l'inclusion de code sur base de la valeur de constantes est fréquemment utilisé pour ajouter des lignes qui ne doivent être exécutées que lorsque l'on veut déboguer un programme. C'est aussi souvent utilisé pour faciliter la portabilité d'un programme entre différentes variantes de Unix, mais cette utilisation sort du cadre de ce cours.

```
#define DEBUG
/* ... */
#ifdef DEBUG
printf("debug : ...");
#endif /* DEBUG */
```

Il est également possible de définir des macros qui prennent un ou plusieurs paramètres **[CPP]**.

Les *headers* standards sont placés dans des répertoires bien connus du système. Sur la plupart des variantes de Unix ils se trouvent dans le répertoire `/usr/include/`. Nous aurons l'occasion d'utiliser régulièrement ces fichiers standards dans le cadre du cours.

Le langage Java a été largement inspiré du langage C et de nombreuses constructions syntaxiques sont similaires en Java et en C. Un grand nombre de mots clés en C ont le même rôle qu'en Java. Les principaux types de données primitifs supportés par le C sont :

- `int` et `long` : utilisés lors de la déclaration d'une variable de type entier
- `char` : utilisé lors de la déclaration d'une variable permettant de stocker un caractère
- `double` et `float` : utilisés lors de la déclaration d'une variable permettant de stocker un nombre représenté en virgule flottante.

Notez que dans les premières versions du langage C, contrairement à Java, il n'y avait pas de type spécifique permettant de représenter un booléen. Dans de nombreux programmes écrits en C, les booléens sont représentés par des entiers et les valeurs booléennes sont définies **[2]** comme suit.

```
#define false   0
#define true    1
```

Les compilateurs récents qui supportent le type booléen permettent de déclarer des variables de type `bool` et contiennent les définitions suivantes **[2]** dans le header standard **stdbool.h** de **[C99]**.

```
#define false   (bool)0
#define true    (bool)1
```

Au-delà des types de données primitifs, Java et C diffèrent et nous aurons l'occasion d'y revenir dans un prochain chapitre. Le langage C n'est pas un langage orienté objet et il n'est donc pas possible de définir d'objet avec des méthodes spécifiques en C. C permet la définition de structures, d'unions et d'énumérations sur lesquelles nous reviendrons.

En Java, les chaînes de caractères sont représentées grâce à l'objet `String`. En C, une chaîne de caractères est représentée sous la forme d'un tableau de caractères dont le dernier élément contient la valeur `\0`. Alors que Java stocke les chaînes de caractères dans un objet avec une indication de leur longueur, en C il n'y a pas de longueur explicite pour les chaînes de caractères mais le caractère `\0` sert de marqueur de fin de chaîne de caractères. Lorsque le langage C a été développé, ce choix semblait pertinent, notamment pour des raisons de performance. Avec le recul, ce choix pose question **[Kamp2011]** et nous y reviendrons lorsque nous aborderons certains problèmes de sécurité.

```
  char string[10];
  string[0] = 'j';
  string[1] = 'a';
  string[2] = 'v';
  string[3] = 'a';
  string[4] = '\0';
  printf("String : %s\n", string);
```

L'exemple ci-dessus illustre l'utilisation d'un tableau de caractères pour stocker une chaîne de caractères. Lors de son exécution, ce fragment de code affiche `String : java` sur la sortie standard. Le caractère spécial `\n` indique un passage à la ligne. **printf(3)** supporte d'autres caractères spéciaux qui sont décrits dans sa page de manuel.

Au niveau des constructions syntaxiques, on retrouve les mêmes boucles et tests en C et en Java :

- test `if (condition) { ... } else { ... }`
- boucle `while (condition) { ... }`
- boucle `do { ... } while (condition);`
- boucle `for (init; condition; incr) { ... }`

En Java, les conditions sont des expressions qui doivent retourner un résultat de type `boolean`. Le langage C est beaucoup plus permissif puisqu'une condition est une expression qui retourne un nombre entier.

La plupart des expressions et conditions en C s'écrivent de la même façon qu'en Java.

Après ce rapide survol du langage C, revenons à notre programme **`src/hello.c`**. Tout programme C doit contenir une fonction nommée `main` dont la signature **[3]** est :

```
int main(int argc, char *argv[])
```

Lorsque le système d'exploitation exécute un programme C compilé, il démarre son exécution par la fonction `main` et passe à cette fonction les arguments fournis en ligne de commande **[4]**. Comme l'utilisateur peut passer un nombre quelconque d'arguments, il faut que le programme puisse déterminer combien d'arguments sont utilisés. En Java, la méthode `main` a comme signature `public static void main(String args[])` et l'attribut `args.length` permet de connaître le nombres de paramètres passés en arguments d'un programme. En C, le nombre de paramètres est passé dans la variable entière `argc` et le tableau de chaînes de caractères `char *argv[]` contient tous les arguments. Le programme **`src/cmdline.c`** illustre comment un programme peut accéder à ses arguments.

```
/*************************************
 * cmdline.c
 *
 * Programme affichant ses arguments
 * sur la sortie standard
 *
 *************************************/

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

int main(int argc, char *argv[])
{
  int i;
  printf("Ce programme a %d argument(s)\n", argc);
  for (i = 0; i < argc; i++)
    printf("argument[%d] : %s\n", i, argv[i]);
  return EXIT_SUCCESS;
}
```

Par convention, en C le premier argument (se trouvant à l'indice `0` du tableau `argv`) est le nom du programme qui a été exécuté par l'utilisateur. Une exécution de ce programme est illustrée ci-dessous.

```
Ce programme a 5 argument(s)
argument[0] : ./cmdline
argument[1] : 1
argument[2] : -list
argument[3] : abcdef
argument[4] : lepl1503
```

Outre le traitement des arguments, une autre différence importante entre Java et C est la valeur de retour de la fonction `main`. En C, la fonction `main` retourne un entier. Cette valeur de retour est passée par le système d'exploitation au programme (typiquement un **shell** ou interpréteur de commandes) qui a demandé l'exécution du programme. Grâce à cette valeur de retour il est possible à un programme d'indiquer s'il s'est exécuté correctement ou non. Par convention, un programme qui s'exécute sous Unix doit retourner `EXIT_SUCCESS` lorsqu'il se termine correctement et `EXIT_FAILURE` en cas d'échec. La plupart des programmes fournis avec un Unix standard respectent cette convention. Dans certains cas, d'autres valeurs de retour non nulles sont utilisées pour fournir plus d'informations sur la raison de l'échec. En pratique, l'échec d'un programme peut être dû aux arguments incorrects fournis par l'utilisateur ou à des fichiers qui sont inaccessibles.

À titre d'illustration, le programme **src/failure.c** est le programme le plus simple qui échoue lors de son exécution.

```
/************************************
 * failure.c
 *
 * Programme minimal qui échoue toujours
 *
 ************************************/

#include <stdlib.h>

int main(int argc, char *argv[])
{
  return EXIT_FAILURE;
}
```

Enfin, le dernier point à mentionner concernant notre programme **src/hello.c** est la fonction `printf`. Cette fonction de la librairie standard se retrouve dans la plupart des programmes écrits en C. Elle permet l'affichage de différentes formes de textes sur la sortie standard. Comme toutes les fonctions de la librairie standard, elle est documentée dans sa page de manuel **printf(3)**. **printf(3)** prend un nombre variable d'arguments. Le premier argument est une chaîne de caractères qui spécifie le format de la chaîne de caractères à afficher. Une présentation détaillée de **printf(3)** prendrait de nombreuses pages. À titre d'exemple, voici un petit programme utilisant **printf(3)**

```
  char weekday[] = "Monday";
  char month[] = "April";
  int day = 1;
  int hour = 12;
  int min = 42;
  char str[] = "SINF1252";
  int i;

  // affichage de la date et l'heure
  printf("%s, %s %d, %d:%d\n", weekday, month, day, hour, min);

  // affichage de la valeur de PI
  printf("PI = %f\n", 4 * atan(1.0));

  // affichage d'un caractère par ligne
  for(i = 0; str[i] != '\0'; i++)
    printf("%c\n", str[i]);
```

Lors de son exécution, ce programme affiche :

```
Monday, April 1, 12:42
PI = 3.141593
S
I
N
F
1
2
5
2
```

Le langage C permet bien entendu la définition de fonctions. Outre la fonction `main` qui doit être présente dans tout programme, le langage C permet la définition de fonctions qui retournent ou non une valeur. En C, comme en Java, une fonction de type `void` ne retourne aucun résultat tandis qu'une fonction de type `int` retournera un entier. Le programme ci-dessous présente deux fonctions simples. La première, `usage` ne retourne aucun résultat. Elle affiche un message d'erreur sur la sortie d'erreur standard et termine le programme via **exit(2)** avec un code de retour indiquant un échec. La seconde, `digit` prend comme argument un caractère et retourne `1` si c'est un chiffre et `0` sinon. Le code de cette fonction peut paraître bizarre à un programmeur habitué à Java. En C, les *char* sont représentés par l'entier qui correspond au caractère dans la table des caractères utilisées (voir **RFC 20** pour une table ASCII simple). Toutes les tables de caractères placent les chiffres `0` à `9` à des positions consécutives. En outre, en C une expression a priori booléenne comme `a < b` est définie comme ayant la valeur `1` si elle est vraie et `0` sinon. Il en va de même pour les expressions qui sont combinées en utilisant `&&` ou `||`. Enfin, les fonctions **getchar(3)** et **putchar(3)** sont des fonctions de la librairie standard qui permettent respectivement de lire (écrire) un caractère sur l'entrée (la sortie) standard.

```
/************************************
 * filterdigit.c
 *
 * Programme qui extrait de l'entrée
 * standard les caractères représentant
 * des chiffres
 ************************************/

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

```c
// retourne vrai si c est un chiffre, faux sinon
// exemple simplifié, voir isdigit dans la librarire standard
// pour une solution complète
int digit(char c)
{
  return ((c >= '0') && (c <= '9'));
}

// affiche un message d'erreur
void usage()
{
  fprintf(stderr, "Ce programme ne prend pas d'argument\n");
  exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
  char c;

  if (argc > 1)
    usage();

  while ((c = getchar()) != EOF) {
    if (digit(c))
            putchar(c);
  }

  return EXIT_SUCCESS;
}
```

Pages de manuel

Les systèmes d'exploitation de la famille Unix contiennent un grand nombre de librairies, d'appels systèmes et d'utilitaires. Toutes ces fonctions et tous ces programmes sont documentés dans des pages de manuel qui sont accessibles via la commande `man`. Les pages de manuel sont organisées en 8 sections.

- Section 1: Utilitaires disponibles pour tous les utilisateurs
- Section 2: Appels systèmes en C
- Section 3: Fonctions de la librairie
- Section 4: Fichiers spéciaux
- Section 5: Formats de fichiers et conventions pour certains types de fichiers
- Section 6: Jeux
- Section 7: Utilitaires de manipulation de fichiers textes
- Section 8: Commandes et procédure de gestion du système

Dans le cadre de ce cours, nous aborderons principalement les fonctionnalités décrites dans les trois premières sections des pages de manuel. L'accès à une page de manuel se fait via la commande `man` avec comme argument le nom de la commande concernée. Vous pouvez par exemple obtenir la page de manuel de `gcc` en tapant `man gcc`. `man` supporte plusieurs paramètres qui sont décrits dans sa page de manuel accessible via `man man`. Dans certains cas, il est nécessaire de spécifier la section du manuel demandée. C'est le cas par exemple pour `printf` qui existe comme utilitaire (section 1) et comme fonction de la librairie (section 3 - accessible via `man 3 printf`).

Outre ces pages de manuel locales, il existe également de nombreux sites web où l'on peut accéder aux pages de manuels de différentes versions de Unix dont notamment :

- les pages de manuel de **Debian GNU/Linux**
- les pages de manuel de **FreeBSD**
- les pages de manuel de **MacOS**

Dans la version en-ligne de ces notes, toutes les références vers un programme Unix, un appel système ou une fonction de la librairie pointent vers la page de manuel Linux correspondante.

Il existe de nombreux livres consacrés au langage C. La référence la plus classique est **[KernighanRitchie1998]**, mais certains éléments commencent à dater. Un tutoriel intéressant a été publié par Brian Kernighan **[Kernighan]**. **[King2008]** propose une présentation plus moderne du langage C.

Footnotes

[1] Si cette option n'était pas spécifiée, le compilateur aurait placé le programme compilé dans le fichier baptisé *a.out*.

[2] *(**1**, **2**)* Formellement, le standard **[C99]** ne définit pas de type `bool` mais un type `_Bool` qui est en pratique renommé en type `bool` dans la plupart des compilateurs. La définition précise et complète se trouve dans **stdbool.h**

[3] Il est également possible d'utiliser dans un programme C une fonction `main` qui ne prend pas d'argument. Sa signature sera alors `int main (void)`.

[4] En pratique, le système d'exploitation passe également les variables d'environnement à la fonction `main`. Nous verrons plus tard comment ces variables d'environnement sont passées du système au programme et comment celui-ci peut y accéder. Sachez cependant que sous certaines variantes de Unix, et

notamment Darwin/MacOS ainsi que sous certaines versions de Windows, le prototype de la fonction `main` inclut explicitement ces variables d'environnement (`int main(int argc, char *argv[], char *envp[])`)