Chapitre 2 : Langage C

Section 2.1 : Le langage C

Le C est un language de programmation écrit dans les années 70 pour le système Unix. Il est *rapide* et permets d'interagir directement avec le matériel.

Un **langage de programmation** permets d'écire des programmes qui seront *compilés* pour être *exécutables* par le processeur (binaires = language Machine). Le **language d'assemblage** quand à lui est un langage proche de la machine pour être le plus rapide possible. (Chaque famille de processeur à son language d'assemblage qui lui est propre. On converti le **langage d'assemblage** en langage Machine via un *assembleur*).

2.1.1 Programme de base

```
    #include <stdio.h>

    int main ( intarg c , char * argv [ ] ){
            printf ( "Hello World!") ; // affiche le message
            return 0;  // fin du prog
    }
```

2.1.2 Compilation

La compilation du fichier hello.c en un exécutable s'effectue sur un système Unix comme suit:

```
    gcc -Wall -o hello hello.c
```

l'argument *-Wall* affiche tous les warnings
l'argument *-o hello* donne le nom de sortie de l'exécutable

2.1.3 Préprocesseur

Au moment de la compilation, le compilateur va exécuter les directives préprocesseur. il s'agit de macros qui sont effectuées en amont de la compilation finale du programme.

- **#define** : permets la définitionn de substitution et est fréquemment utilisé pour définir des constantes qui sont valables dans l'ensemble du programme.
- **#include** : macro permettant l'inclusion de fichiers.
  - < *stdio.h* > : librairie de fonctions standard permettant d'interragir avec les entrées et sorties standard et notament *printf*
  - < *stdlib.h* > : fonction et constantes de la librairie standard

```
    #include <...>
    #define ZERO 0 // remplace ZERO par 0
```

On peut les utiliser pour :
* la portabilité de l'app * diminuer la taille des identifiants * redéfinir des pointeurs (attention car un ptr reste un ptr)

2.1.4 Strings

Les strings sont des tableaux de caractères. en C, ils se terminent par la valeur '\0'

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

- %c fait référence à une variable de type char
- %d fait référence à une variable de type int
- %s fait référence à la variable string qui est un string

2.1.5 Constructons Syntaxiques

```
    if (COND){...}else{...}
    while(COND){...} //if cond false do nothing
    do {...} while(COND); //if cond is false then don't repeat
    for(INIT;COND;INCR){...}
```

2.1.6 Les arguments d'un programme

Pour indiquer si un programme s'est terminé avec ou sans erreur (0 : Succes ; 1 : Error). Mais un autre code peut être renvoyé pour donner plus d'informations sur l'erreur.

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

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

2.1.7 Le Manuel

On peut accéder au manuel via la commande **man**

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

Section 2.2 : Types de données

Les types de données et leur représentation en mémoire:

- decimal : 123
- Binaire : **0b** 1111011
- Octal : **0** 173
- Hexadécimal :**0x** 7B

On peut obtenir la taille en mémoire d'un type de données avec :
`sizeof(DATA_TYPE)`

2.2.1 Les nombres entiers

Les **nombres signés** sont représentés sous la forme : Signe (négatif si = 1) - Nombre.
Les **nombres non-signés** sont représentés sous la forme binaire std $5 = 2^2 + 2^0$.

2.2.2 IEEE 754

Le **Standard IEEE 754** est une représentation des nombres réels sous forme : Signe; exposant; signifiant. Il existe la single et la double précision (respectivement 32 et 64 bits)

2.2.3 Les tableaux

Dans les anciennes version du langage langage C, les tableaux étaient de taille fixe. **Attention** pas de Tab.length en C il faut donc prévoir de garder la taille du tableau en mémoire si on veut l'utiliser par la suite

```
    #define N 10
    float vecteur[N]
    float matrice[N][N]
    int tab[3] = {1,2,3};
```

attention pas de buffer overflow car pas d'exception

2.2.4 Caractères et chaines de caractères

Le langage C n'integre pas d'office les boolean et les strings. En C, les strings sont des tableaux de caractères dont le dernier élément contient la valeur '\0'.

```
    char string[20]= "text";
    printf("%s \n",string);
```

Les lettres étant des char (ASCII de 7 ou 8 bits) stockés sous la forme d'entiers, on peut donc effectuer des manipulations numériques avec des char.

Encodage	Particularité
ASCII	caractère Anglais
ISO8859	Latin avec Accents
UNICODE	Tous les caractères dans toutes les langues

Il n'existe pas de mécanisme d'exception en C. Celà pose des problèmes sécurité & des possibilités de Buffer Overflow. Une chaine de caractères se termine toujours par un '\0' (equivalent à 0)

2.2.5 Les Pointeurs

En C, le programmeur peut interragir directement avec la mémoire où les données qu'un programme manipule sont stockés.

La **mémoire** est une zone qui est définie et accessible via son adresse. Un **pointeur** est une variable contenant l'adresse d'une autre variable.

```
    &var // adresse à laquelle une variable est stockée
    var // variable en mémoire
```

```
    *ptr // récupère la valeur à l'adresse du pointeur
```

En C, contrairement au java. Il n'y a pas de garbage collection qui retire de la mémoire les objets qui ne sont plus utilisés. Il faut donc vider la mémoire qui n'est plus utilisée.

2.2.6 Les structures

En C, contrairement au java (et autres langages orientés objet) on ne peut pas créer d'objets mais on peut créer des types de données (appelés structures). Les Structures n'ont pas de méthodes liés via l'encapsulation dans la classe.

Une **structure** est une combinaison de différents types de données simples ou structurés.
Dans les premières version du langage, les structures avaient une taille fixe

- **Créer une structure**

```
    struct NOM-STRUCTURE {
        int VARIABLE1;
        int VARIABLE2;
    };
```

- **Créer une instance de la structure**
  `struct NOM-STRUCTURE NOM-INSTANCE = {1,2};`
- **Accéder à une variable en mémoire directement**
  `NOM-INSTANCE.VARIABLE1= 2;`
- **Accéder à l'élément d'une structure contenue dans un pointeur**
  `(* ptr).VARIABLE1 ptr->VARIABLE1`

2.2.7 Les Alias

On peut redéfinir des noms de structures : `typedef int ENTIER;`

2.2.8 Les fonctions

Les fonctions sont des découpes simples de tâches complexes. On peut définir une fonction comme suit :

```
    type_de_retour nom_fonction(type_var_1 nom_var_1, type_var_

    void hello(){...} //fonction sans arguments et sans valeur

    int hola(int age){...} //fonction avec argument qui retourn
```

```
int main(int argc, char *argv[]){...}// fonction main
```

La fonction **main** est la **fonction principale** et **obligatoire** d'un programme.

Déclaration : Indique au compilateur le type des arguments et le type de la valeur de retour(en cas de fonctions). Toutes les fonctions et variables doivent être déclarées avant d'êtres utilisées.

Définition : Le corps de la fonction est spécifiée dans la déclaration ou dans le cas d'une variable déans son initialisation.

En C les fonctions et les pointeurs peuvent être utilisés en argument.

2.2.9 Manipulation bits

On utilise souvent ces expressions avec des représentations non signées (unsigned char ou unsigned int).

```
r = ~a;  //négation bit à bit
r = a & b; // conjonction bit à bit
r = a | b; // disjonction bit à bit
r = a ^ b; // xor bit à bit
a = n >> B // décale les bits représentant de n de B bits v
```

Pour forcer des bits dans un unsigned char, on peut utiliser la méthode suivante :

```
r = c & 0x7E; // force les bits de poids faibe et forts a 0
r = d | 0x18; // force les bits 4 et 3 à 1 (00011000)
```

Xor peut être utilisé pour le chiffrement et le déchiffrement :
`(A XOR B ) XOR B = A`

IL est important de ne pas confondre les expressions logiques (|| et &&) avec les opérateurs binaires (| et &).
info : Les expressions logiques sont évaluées de G à D. On peut donc les utiliser de la façon suivante `((ptr != NULL) && (ptr->den>0))` . Car la partie de droite ne sera pas exécutée si ptr est NULL.

Section 2.3 : Declarations

Les variables sont définies par leur portée.
La portée d'une variable : partie du programme ou la variable est accessible et où sa valaur peut-être modifiée.

Les variables globales sont des variables accessibles de partout dans le

programme. Elles doivent être utilisées de façon parcimonieuses (utilisation mémoire plus importante). Pour les variables locales, les premières versions du langage C imposaient leur définition au début des blocs.

Dans les premières versions de C on devait définir les variables au début de chaque bloc.

Pour définir des constantes on peut :

```
    #define M_PI 3.14159265
    const double pi=3.14159265
    const struct fract{int num;int den;}demi={1,2};
```

Section 2.4 : Unions et énumérations

2.4.1 énumérations

enum est utilisé pour définir un type de données énumérées. càd un nombre fixe de valeurs possibles. (val stockées sous la forme d'entiers)

```
    typedef enum{
        monday, tuesday, wednesday, thursday, friday, saturnday
    }day;
    day jour = monday;
```

2.4.2 Unions

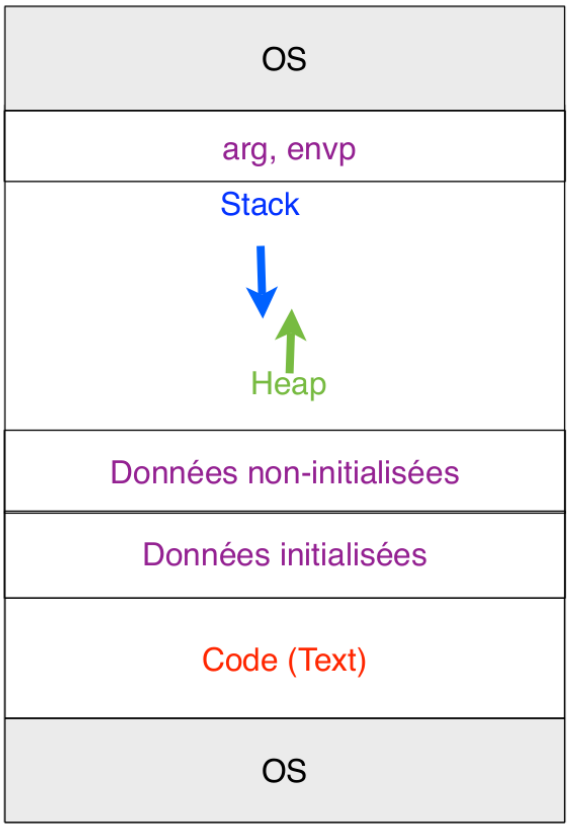
union permet de réserver une zone en mémoire pour stocker plusieurs types de variables possibles

```
    union u_t{
        int i;
        char c;
    }u;
    u.i = 12;
```

u peut ici contenir **soit** un *int*, **soit** un *char*, mais pas les deux en même temps comme dans le cas d'une structure.

Section 2.5 : L'organisation de la mémoire

La mémoire peut-être divisée en six zones principales :



2.5.1 Le segment text

Contient toutes les **instruction** qui sont exécutées par le microprocesseur. (uniquement accessible en lecture)

2.5.2 Le segment des données initialisées

Contient l'ensemble des données et chaînes de caractères qui sont utilisées dans le programme. (il comprend l'ensemble des variables globales déjà initialisées)

2.5.3 Le segment des données non-initialisées

Contient les valeurs des variables non-globales ou les variables globales non initialisées. Elle est initialisée à 0 lors du démarage du programme. Attention toute fois aux variables locales qui ne sont pas explicitement intialisées

2.5.4 Le tas (ou heap)

C'est dans une des 2 zones dans laquelle un programme peur obtenir de la mémoire supplémentaire pour y stocker de l'information. Un programmeur peut réserver une zone permettant de stocker des données et y associer un pointeur. (brk(2) et sbrk(2) modifient la taille du heap). On associe ainsi un pointeur à

l'adresse réservée par le programme.

Malloc

En pratique, on utilise malloc(3) pour allouer de la mémoire et free(3) pour la libérer ce qui a été aloué manuellement en mémoire (variables). Attention l'oubli de ces libérations mémoires peuvent mener à des Memory leaks.

Malloc, contrairement à calloc ne réinitialise pas la zone mémoire libérée.
Malloc retourne un (void*) qu'il faut ensuite caster.

```
    #include <sdtlib.h>
    string= (char * ) malloc(length * sizeof(char))
    free(string);
```

Calloc

```
void *calloc(size_t num_element, size_t size); // base
char *ptr = calloc(15,size(char));//exemple
```

2.5.5 La pile (ou stack)

Cette zone est très importante, elle stocke :

- l'ensemble des variables locales
- les variables de retour de toutes les fonction qui sont appelées
- Les arguments placés aux fonctions

Cette zone tire son nom de son mode de gestion à la manière d'une pile.

Les arguments et variables d'environnement

- argc : nombre d'arguments
- char* argv[] : les arguments
- argv[0]: nom du programme exécuté

Les **variables d'environnement** sont toutes les variables permettant d'accéder à certaines informations de l'environnement qui lance le programme. (ex : path , lang, shell , home, ...).

Section 2.6: Compléments de C

2.6.1 Pointeurs

On peut utiliser des pointeurs vers :

- des données primitives
- des structures
- n'importe quel information dans un programme C
  cfr pointeurs et malloc

2.6.2 De grands programmes en C

programme principal

```
    #include "bob. h"
```

Fichier header

```
    #ifndef _BOB_H_
    #define _BOB_H_
    int funct (float, int);
    #endif
```

fichier c complémentaire

```
    #include "bob.h"
    int funct(float a, int b) { ...}
```

Makefile

```
    myprog: main.o min.o
        gcc -std=c99 -o myprog main.o min.o

    main.o: main.c min.h
        gcc -std=c99 -c main.c

    min.o: min.c min.h
        gcc -std=c99 -c min.c
```

une variable ou fonction definie comme static n'est accessible qu'aux entitées de ce module