Programmation

Langage C

L2 Mathématique et Informatique

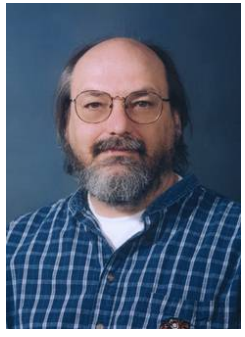
Université de Marne-la-Vallée

Objectifs de ce cours

- apprendre le langage C
- produire du code propre
- savoir manipuler des adresses pour comprendre les opérations se déroulant en mémoire
- comprendre les forces et les faiblesses de ce langage, prendre du recul pour mieux appréhender les autres langages et paradigmes

Historique `Unix et C`

- 1969 : Naissance du système d'exploitation `Unix` (Ken Thompson et Dennis Ritchie) écrit en assembleur
- 1970 : K. Thompson et D. Ritchie dessinent un nouveau langage pour réécrire `Unix`
- 1972 : laboratoire Bell, D. Ritchie et K. Thompson
- 1989 : Norme ANSI du langage C
- 1991 : Linus Torvalds, étudiant finlandais, propose une réécriture libre du noyau `Unix` : `Linux`
- 2011,norme C11, formellement ISO/CEI 9899:201



Thompson

Ritchie

Torvalds

Pourquoi le C

- proche de la machine, il oblige à se poser des questions sur le déroulement
- syntaxe relativement simple, reprise par de nombreux langages plus modernes
- executables rapides
- encore très répandu, portable
- python est (*partiellement*) écrit en C
- programmation système (noyau Linux)

Différences avec python

- Le C est compilé:
  pas d'interpréteur permettant de tester des instructions. il faut écrire un programme complet et le traduire en un fichier exécutable à l'aide d'un logiciel de compilation `gcc`:
  : `gcc projet.c -o monjeu` traduit le fichier `projet.c` contenant le code en C en un fichier en langage machine de nom `monjeu`.
- les instructions simples se terminent par un `;`
  `a=a+1;`
- les blocs doivent être explicitement indiqués à l'aide d'accolades: { et }. Pour la lisibilité conservez les habitudes d'indentation acquises avec python;
- avant d'être utilisé tout élément doit avoir été déclaré.
  La déclaration permet au compilateur de fixer le type d'une variable et de lui associer un emplacement en mémoire:
  `int a;` indique que `a` désignera dans la suite une variable de type entier (dont la valeur n'a pas été fixée).

Les fichiers sources

- Le code en C d'un programme doit être contenu dans un fichier dont le nom se termine par `.c` (nous verrons plut tard que l'interface peut être précisée dans un fichier d'extension `.h`)
- syntaxe très précise à respecter
- séparateurs = {espace, tabulation, retour ligne}
- programmation modulaire : un fichier par thème (plus tard)
  - bibliothèque mathématique
  - interface graphique
  - entrées/sorties (`<stdio.h>`)
  - ...

Le premier programme

fichier `hello.c` :

```
1  /* Affichage du message Hello world! */
2
3  #include <stdio.h>
4
5  int main(void){
6     printf("Hello world!\n");
7     return 0;
8  }
```

- On met les commentaires entre `/* ... */`. Les commentaires de fin de ligne avec double slash ne passent pas la norme `ansi`!
- Les commandes commençant par # sont des directives pour le préprocesseur. Elle sont exécutées lors d'une première phase de traduction. Ici on demande au compilateur l'inclusion de la bibliothèque standart du C `stdio.h` permettant l'utilisation de la fonction `printf`.

Le premier programme

fichier `hello.c` :

```
1  /* Affichage du message Hello world! */
2
3  #include <stdio.h>
4
5  int main(void){
6     printf("Hello world!\n");
7     return 0;
8  }
```

- `main` est le point d'entrée de l' exécutable. C'est la fonction qui sera exécutée au lancement du programme.
- Le type de retour apparaît en premier dans le prototype de la fonction. La fonction `main` retourne ici un entier `int`.
- Les paramètres de la fonction `main` sont ensuite précisés. Ici la fonction ne reçoit rien ce qu'on indique par `void`
- La valeur de l'expression située après le mot clé `return` est retournée par la fonction.. Par convention, la fonction `main` utilise 0 pour indiquer un déroulement correct.
  `return ;` signifie "retourner rien", ceci est correct avec un type de retour `void`.

Le premier programme

fichier `hello.c` :

```
1  /* Affichage du message Hello world! */
2
3  #include <stdio.h>
4
5  int main(void){
6     printf("Hello world!\n");
7     return 0;
8  }
```

- L'accolade ouvrante { débute un bloc d'instructions. Ce bloc se ferme avec une accolade fermante }. Les blocs peuvent s'imbriquer.
- La ligne 6 du programme affiche à l'écran la chaîne de caractère souhaitée.

Le premier programme

- **Compilation :**
  `gcc hello.c -o Helloworld`
- **Exécution :**
  `./Helloworld`

`gcc` est le programme de compilation sous `Linux`. L'option `-o` de `gcc` demande la compilation en précisant le nom à donner à l'exécutable (ici `Helloworld`)

Quelques options courantes de `gcc`

- `-ansi` : compilation de C à la norme ANSI
- `-Wall` : Mise en oeuvre de certains warnings très utiles (mais pas tous)
- `-E` : préprocesseur uniquement

Même s'ils sont parfois durs à lire, les messages d'erreur du compilateur doivent être pris en compte. Lorsqu'ils sont nombreux, corriger les premières erreurs puis recompiler.

Autre exemple

```c
#include <stdio.h>
#define PI 3.14      /* une constante */

/* variables globales BEURK */
float a,b;       /*deux decimaux*/

float sum(float x, float y){     /* une fonction */
  return x+y;
}

int main(void){
  float c,d;
  a = PI;
  b = 1;
  c = sum(a,b);
  printf("%f + %f = %f\n", a,b,c);
  printf("d = %f", d); /* d n'est pas initialisee !!! */
  return 0;
}
```

Le C permet l'utilisation d'une variable non initialisée. Ce fait est à considérer comme une "faiblesse" du langage (des flags de `gcc` peuvent détecter ces problèmes).

Les types numériques

Le type permet de préciser au compilateur la taille nécessaire pour représenter une donnée et la manière de la représenter (il s'agit dans tous les cas de suites de 0 et de 1). En python, il est déterminé à l'utilisation alors qu'en C, c'est le programmeur qui doit décider.

1. Les types entiers : ils désignent des intervalles finis de l'ensemble des entiers relatifs.
   - le type `int` pour les entiers compris entre $-2147483648 (= -2^{31})$ et $2147483647 (= 2^{31} - 1)$. Les `int` sont représentés avec 4 octets.
   - le type `unsigned int` pour des entiers positifs compris entre 0 et $4294967295 (= 2^{32} - 1)$
   - le type `char` et le type `unsigned char` pour les caractères (identifiés à leur code numétique): `'a'+1` vaut `'b'`.

   Attention aux débordements avec un type signé, la somme de 2 `int` positifs peut donner une valeur négative.

2 Les types décimaux :
ils désignent des sous ensembles de décimaux avec un précision donnée

- `float` simple précision
- `double` double précision
- `long double` grande précision

Attention aux erreurs d'arrondis on peut avoir $a + b == a$ si a est beaucoup plus grand que b

```
1    float a,b,c;
2    a= 10000;
3    b=0.00001;
4    c=a+b;
5    printf("%f + %f = %f \n",c);
```

affiche $10000.000000 + 0.000010 = 10000.000000$

Les opérateurs

Les opérateurs $+, -, *, /$ s'appliquent uniquement à des types numériques (pas de mélange `int` et `string`). La division dépend du type de ses opérandes: si les deux opérandes sont de type `int`, il s'agit de la division entière ($3/2$ vaut 1); si au moins une opération est d'un type décimal, il s'agit de la division décimale ($3.0/2$ vaut1.5).

Les variables

- syntaxe identique à celle de python
- elle doivent être explicitement déclarées avec leur type
- une bonne habitude est de les déclarer au début du bloc où on les utilise:

```
1  int main(void){
2     int a,b;
3     double x;
4  ....
5  }
```

Une variable désigne directement un emplacement en mémoire (*une adresse*) où est stockée la valeur. C'est le type de la variable qui permet de déterminer la taille de l'emplacement et la manière de comprendre la suite de bits qu'on y trouve. Le type est fixé par le compilateur.
L'adresse d'une variable se récupère avec l'opérateur &.

Les instructions simples

- instructions de déclarations, précisent le statut d'éléments utilisés ensuite,
- instructions basées sur une expression, correspondent à "évaluer l'expression". Si l'expression n'a pas d'effet de bord (*action*) l'instruction ne sert à rien

```
1  int a;/* declaration */
2  float x=3.14;/* declaration avec affectation */
3  f(x);/* valeur perdue si f renvoie une valeur ! */
4  a=4;/* l'affectation est une expression */
5       /* effet de bord a vaut 4 */
```

- instruction vide :

```
1  ;
```

déconseillée, peut apparaitre dans certaines constructions .

Les entrées sorties formatées

Elles s'effectuent grâce aux fonctions de la bibliothèque standard. Nous utiliserons dans un premier temps

- `printf()` pour l'affichage
- `scanf()` pour la lecture.

il faut indiquer grâce au format le nombre et le type des valeurs à traiter. Les precisions de format sont formées du caractère `%` suivi d'une ou plusieurs lettres. Les principaux sont:

- `%d` pour un nombre entier
- `%f` pour un nombre réel `float` ou `double`
- `%c` pour un caractère

printf()

```
1  printf("valeur=%d \n",x);
```

affiche la valeur de la variable `x` et passe à la ligne. Le premier argument est obligatoirement une chaine de caractère. Tout ce qui n'est pas une précision de format sera affiché sans changement. Les précisions de format sont remplacées par l'évaluation de l'expression correspondante dans la suite des arguments. Bien entendu le nombre et le type des arguments doivent correspondre.

```
1  int a,b;
2  a=3;b=5;
3  printf("le produit de %d et %d est %d,", a,b,a*b);
4  printf("le quotient est %f \n",a/b);
```

scanf()

```
1 scanf("%d",&x);
```

lit un entier au clavier et place la valeur à l'adresse de x. Le premier argument est obligatoirement une chaine de caractère. Dans un premier temps, ne mettez qu'une seule précision de format dans cette chaine. Pour les types numériques les séparateurs `espace, passage à la ligne , tabulation` sont ignorés (on peut en mettre autant que l'on veut).Il faut fournir à `scanf` l'adresse de la variable qui reçoit la valeur lue au clavier. Pour un type simple on utilise &

```
1 int a,b;
2 printf("entrez 2 entiers");
3 scanf("%d",&a);
4 scanf("%d",&b);
```

Instructions conditionnelles
if

```
1  if (condition)
2     instruction
```

La condition est evaluée et l'instruction est effectuée si la condition est vraie La condition, placée entre parenthèses, se construit à l'aide:
des comparateurs `==, !=, <,>,<=,>=`
des connecteurs logiques `||` pour OU, `&&` pour ET `!` pour NON.
L'évaluation est paresseuse.
S'il y a plusieurs instructions à effectuer, on utilise un bloc

if...else

```
1  if(condition)
2     instructionV
3  else
4     instructionF
```

La condition est evaluée. Si la condition est vraie l'instructionV est effectuée sinon l'instructionV est effectuée. le `else` se rapporte au premier `if` libre précédent. Utiliser des accolades pour qu'un `if` ne soit plus libre.

```
1  if(a%2==0){
2       if  (a>10)
3           printf("%d est pair et superieur a 10\n");
4     }
5  else
6     printf("%d est impair\n);
```

switch

Pas d'instruction `elif`. L'instruction d'aiguillage `switch` peut-être utilisée si les comparaisons portent sur des valeurs de type entier (`int` ou `char`)

```
1 switch(variable){
2   case valeur1: instructions
3   case valeur2: instructions
4   ...
5   case valeurn: instructions
6   default: instructions/* optionnel*/
7 }
```

`valeur1` `valeur2`... `valeurn` (les *étiquettes)* doivent être des constantes, définies avant compilation. La valeur de la variable est comparé aux *étiquettes* :
si une égalité est trouvé le programme se poursuit à partir de la ligne de cette étiquette;
sinon lorsqu'une instruction `default` est présente le programme se poursuit à partir cette ligne.

```
1       printf(``un entier :'')
2       scanf("%d", &i);
3       switch(i){
4         case 1 : printf("␣1");
5         case 2 : printf("␣2");
6         case 3 : printf("␣3");
7         case 4 : printf("␣4");
8         default : printf("␣coucou␣!\n");
9       }
```

si on entre 2

```
un entier :2
 2 3 4 coucou !
```

si on entre 5

```
un entier :5
 coucou !
```

break

`break` permet de sortir imédiatement du bloc et donc de ne pas effectuerles instructions suivantes du bloc.

```
1       printf("un entier :")
2       scanf("%d", &i);
3       switch(i){
4         case 1 :
5         case 2 : printf(" 1 ou 2\n");break;
6         case 3 : printf(" 3\n");break;
7         default : printf(" inconnu!\n");
8       }
```

si on entre 2

```
 un entier :2
  1 ou 2
```

boucles

1. **boucle for**

```
1   for(initialisation;condition;incrementation)
2       instruction
```

`initialisation` est une expression évaluée en premier,une seule fois
`condition` est évaluée
si elle est vraie
    `instruction` est effectuée
    `incrémentation` est evaluée.
    on recommence à l'évaluation de `condition`
si elle est fausse on sort de la boucle.
A préférer quand la variable de boucle évolue toujours de la même manière

```
1   int i,somme;
2   ...
3   somme=0;
4   for(i=0;i<n;i=i+1)
5       somme=somme+i;
```

Si la boucle porte sur plusieurs instructions on utilise un bloc

```
1   int i,somme,valeur;
2   ...
3   for(i=0,somme=0;i<n;i=i+1){
4       printf("entrez un entier");
5       scanf("%d",&valeur);
6       somme=somme+valeur;
7   }
```

Seuls les deux ';' sont obligatoires.
Si la condition est omise, elle est considérée vraie. On obtient une bocle infinie dont il faudra sortir par exemple avec un `break` ou un `return`.

while

```
1  while(condition)
2      instruction
```

On évalue la condition, tant qu'elle est vraie, on effectue l'instruction et on recommence l'évaluation de la condition.
A réserver plutot lorsque la variable sur laquelle porte la condition n'évolue pas toujours de la même manière

```
1  while(terme!=1){
2      nbetapes=nbetapes+1;
3      if(terme%2==0)
4          terme=terme/2;
5      else
6          terme=3*terme+1;
7  }
```

do...while

```
1 do
2     instruction
3 while(condition);
```

l'instruction est effectuée au moins une fois, avant que la condition ne soit évaluée.

```
1 do{
2     printf"entrez un entier positif :");
3     scanf("%d",&valeur);
4 }while(valeur<0);
```

A utiliser avec méfiance...

Code propre

- indentation claire et constante
- commentaires dans les zones critiques
- nom de variables, structures et fonctions avec du sens et formalisés de manière unifiée dans tout le code (`nom_de_fonction`, `NomDeFonction`, `Nom_De_Fonction`, ...)
- code lisible par tous
- éviter le mode gourou ou obscurantiste (du moins si vous voulez être lu...)
- fixer et tenir des standards de développement :
  - entête de documentation avec référence de l'auteur,
  - fonctions documentées,
  - disposition du code, sauts de ligne réguliers, ...

Aide sous `Unix`

Le manuel `man` d'`Unix` constitue le juge de paix! Tout y est consigné.
La politique de développement d'`Unix` impose que toutes les nouvelles fonctionnalités soient documentées. Le manuel constitue ainsi un exemple sérieux de modélisation de spécifications.
Pour chaque fonction :

- NAME : nom de la (ou des) fonction(s)
- SYNOPSIS : prototype(s) et bibliothèque(s)
- DESCRIPTION : description comportementale
- RETURN VALUE : description de la valeur de retour
- NOTES : toutes informations méritantes d'être consignées
- BUGS : bogues connus et confirmés
- EXAMPLES : exemples d'utilisation
- SEE ALSO : fonctionnalités similaires
- COLOPHON : informations sur la dernière révision

Section 3 du manuel `Unix`

La section 3 du manuel d'`Unix` est dédiée à la programmation en langage C. Pour toute précision sur une fonction `foo` du langage C (et des bibliothèque standards), il suffit ainsi de taper `man 3 foo` dans un terminal.
exemple : `man 3 qsort`

```
QSORT(3)            Linux Programmer's Manual                    QSORT(3)

NAME
       qsort, qsort_r - sort an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));

       void qsort_r(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *, void *),
                  void *arg);
....
```