Chapitre 2

Langage C

2.1 Le langage C, un langage non interprété

Le C est un langage de programmation comme il en existe beaucoup d'autres (C++, Fortran, Python, Matlab, IDL, ADA...). Il existe deux grandes familles de langages :

- Les langages **interprétés** : lorsqu'on exécute des programmes écrits en langage interprété, les instructions de code sont converties en instructions machine en temps réel et sont directement exécutées. `Matlab` est un exemple de langage interprété.
- Les langages **non-interprétés** : pour des programmes écrits dans de tels langages, ces deux phases sont explicitement dissociées. La langage C est un exemple de langage non-interprété. L'exécution d'un programme C requiert donc 3 étapes successives : l'écriture, la compilation, et l'exécution.

2.1.1 L'écriture du programme

Un programme C est écrit dans un fichier ascii (texte) à l'aide d'un éditeur quelconque (kwrite par exemple). Par convention, on utilise l'extension `.c` pour le nom des fichiers texte qui contiennent du code C, par exemple `fichier.c`.

Voici un exemple de programme simple (il contient 2 lignes de code !) :

```
#include<stdio.h>
main(){printf("Bonjour\n");}
```

Mise en pratique :

- Écrire un tel programme dans un fichier de nom `prog.c`. Bien penser à sauver ce fichier.

2.1.2 La compilation

Dans les langages non-interprétés, la compilation permet de convertir le code écrit en C en des instruction en langage machine que l'ordinateur sera capable d'exécuter. Cette opération s'effectue grâce à un programme particulier que l'on appelle un **compilateur**. Lorsqu'il est appelé, celui-ci lit toutes les lignes du code, les interprète et produit un fichier **exécutable**, qui contient les instructions machine et que l'ordinateur pourra ensuite exécuter.

Il existe souvent plusieurs compilateurs disponibles (des gratuits, d'autres payants). Ici, nous utiliserons le compilateur **gcc** (pour *GNU C Compiler*). De l'information et de l'aide sur l'utilisation de ce compilateur peuvent être obtenus par les commandes `>man gcc`, `>gcc --help`, ou `>gcc --help -v`. Il existe de très (très, très) nombreuses options de compilations qui permettent au compilateur de traduire différemment les lignes de code en langage machine. Le plus souvent, elle servent au debuggage, et à l'optimisation des codes.

La compilation s'effectue depuis un terminal en appelant le compilateur suivi du nom du programme : `>gcc prog.c` (il faut pour cela être dans le répertoire qui contient le programme à compiler). Lorsque la compilation arrive au bout, elle produit par défaut un fichier exécutable de nom de `a.out`. Pour distinguer les exécutables provenant de programmes différents, on spécifie le plus souvent le nom qui l'on souhaite lui donner avec l'option de compilation `-o`. Ainsi pour spécifier le nom d'exécutable `prog.x` on utilise la syntaxe suivantes :

```
>gcc prog.c -o prog.x
```

De très nombreuses erreurs de code peuvent être détectées à cette étape de compilation. Il peut s'agir de *warnings* (le compilateur suspecte un problème) et d'*errors* (le compilateur n'arrive pas à interpréter les lignes de code). Lorsque de tels problèmes sont rencontrés, le compilateur donne en général des explications sur l'origine du problème, ainsi que sur le numéro de ligne du programme qui pose problème. Ces informations ne sont pas toujours très explicites, mais elle sont souvent **extrêmement utiles** pour debugguer un code.

Mise en pratique :
- compiler le programme `prog.c`
- l'éditer et y insérer une erreur (par exemple retirer le `";"`, ou une `"}"`). Re-compiler.
- corriger l'erreur et re-compiler.

2.1.3 L'exécution

Une fois un exécutable `prog.x` créé, on peut le lancer autant de fois que l'on veut, depuis le répertoire qui contient cet exécutable, avec la syntaxe suivante :

```
>./prog.x
```

Mise en pratique :
- exécuter le programme `prog.x`

Lorsque l'on édite un programme pour y faire des modifications, il faut impérativement répéter ces 3 étapes pour les mettre en pratique :
- Sauver les changements
- Compiler
- Exécuter

Pour ce faire, le rappel des commande du shell avec ↑ est bien pratique...

2.2 Structure générale d'un programme et règles globales

Après analyse du programme simple donné plus haut, on peut noter un certain nombre de généralités valides pour tous les programmes écrits en C. En particulier, un programme se structure toujours de la même manière :

```
#include<lib1.h>
#include<lib2.h>
...
main(){
    instruction;
    instruction;
    ...
    instruction;
}
```

De plus, il doit se plier aux règles suivantes :

- Les lignes `#include<lib.h>` permettent de charger des bibliothèques de fonctions prédéfinies dont le programme peut avoir besoin.
- Le programme principal s'appelle toujours `main` et n'a (en général) pas d'arguments.
- L'ensemble des instructions du programme sont données entre `{}`.
- Chaque instruction se termine par un `;`.
- On peut rajouter autant d'espaces, de tabulations ou de retours à la lignes que l'on désire (ils ne sont pas interprétés), ce qui permet d'aider à la lisibilité (**utilisez-les abondamment pour structurer vos programmes !**).
- On peut rajouter une ligne de commentaires (non interprétée) en la commençant par les symboles `//` de la manière suivante : `//blablabla`.
- On peut rajouter des commentaires qui peuvent couvrir plusieurs lignes entre les symboles `/*` et `*/` de la manière suivante : `/*blablabla*/` (**utilisez-les abondamment pour aider à la lisibilité et au debuggage !**)
- Les minuscules et majuscules sont interprétées différemment (par exemple les variables `x` et `X` sont différentes).
- Les éventuelles fonctions sont définies en dehors du programme principal.

2.3 Commandes de base

2.3.1 Affectation

Les affectations de variables s'effectuent intuitivement avec le signe `=`. Les valeurs d'un tableau doivent être affectées une par une en utilisant les crochets `[]` (attention, en C, les indices commencent à 0). Ainsi par exemple :

```
x=2; y=5.9;
z=x;
tableau[7]=2.5;
matrice[3][1] = 5.2;
```

Comme toute autre instruction C, les affectations doivent se terminer par un point virgule `;`.

Mise en pratique :

- Essayer d'écrire un programme simple qui attribue la valeur `2` à une variable `x`.

2.3.2 Déclaration

Avant toute affectation ou utilisation, chaque variable doit être *déclarée*. Lors de cette opération, on précise au compilateur qu'il va falloir mettre de côté l'espace mémoire nécessaire pour stocker la valeur de cette variable. Les variables de *types* différents (entiers, réels...) n'occupant pas le même espace mémoire, il faut impérativement préciser ce type à chaque déclaration. La syntaxe standard est donc : `type variable;`. Les différents types sont donnés dans le tableau 2.1. Lors de leur déclaration, la taille des tableaux doit être

type	syntaxe
entier par défaut	`int`
entier court (16 bits, $\lesssim 3 \times 10^4$)	`short int`
entier long (32 octets, $\lesssim 2 \times 10^9$)	`long int`
entier très long (64 bits, $\lesssim 9 \times 10^{18}$)	`long long int`
réel simple précision (32 bits, ≈6 chiffres)	`float`
réel double précision (64 bits, ≈15 chiffres)	`double`
réel triple précision (80 bits, ≈17 chiffres)	`long double`
chaîne de caractère (1 seul caractère)	`char`
vide (utile pour certaines fonctions)	`void`

TABLE 2.1 – *Les différents types C et la syntaxe de leur déclaration. Le type "entier par défaut" correspond à l'un des 3 types d'entiers possibles, selon les machines (souvent le type "entier long"). On peut spécifier que des entiers et réels sont strictement positifs en rajoutant "unsigned" juste avant le type. Par exemple : "unsigned long long int". Une chaîne de caractères doit être déclarée comme un tableau de caractères.*

spécifiée avec des crochets `[]` de la manière suivante : `type tableau[taille];`. La déclaration peut aussi être l'occasion d'affecter une variable avec une valeur d'initialisation. Enfin, plusieurs variables de même type peuvent être déclarées ensemble. Ainsi par exemple :

```
long int n;
int i=10, j, k=5;
double x=5.4, y, z;
int tableau_d_entiers[25];
double tableau_de_reels[25];
double matrice_de_reels[4][4];
char mot[10], mot2[]="hello";
```

Mise en pratique :

– Modifier le programme précédent en incluant la déclaration de la variable `x`.

2.3.3 Opérations

La plupart des opérations se font de manière intuitive avec les opérateurs usuels (bien que quelques éléments de syntaxe puissent différer notablement d'autres langages). Comme toutes les instructions, elles doivent être terminées par un point virgule `;`.

- Les opérations simples : `+,-,*,/` se font naturellement.
  Par exemple : `z=x*y;` Attention, si les deux variables soumises à ces opérations sont de même type (par

exemple des entiers), alors le résultat est aussi de ce type. Ainsi, si $i = 1$ et $j = 3$ sont deux variables déclarées comme entiers, alors le résultat de l'opération `i/j;` est aussi un entier ici $i/j = 1/3 = 0$.
- Le reste de la division de i par j se calcule avec la syntaxe `i%j`.
- Les fonctions mathématiques de base sont incluses dans la bibliothèque de mathématiques (il faut ajouter `#include<math.h>`). Ainsi, `exp()`, `sin()`, `cos()`, `fabs()`, sqrt()...
- L'élévation à une puissance se fait avec l'appel à une fonction spécifique : `y=pow(x,n);` élève x à la puissance n. En particulier, les symboles `^` et `**` ne sont pas interprétés.
- Les conversions de type se font avec les syntaxes données dans la table 2.1. Ainsi par exemple : `n = (int) 5.3;` arrondit le nombre décimal et attribue la valeur 5 à la variable n.
- Certaines constantes sont prédéfinies comme par exemple `M_PI` pour le nombre π.
- Certains raccourcis existent pour des opérations classiques comme par exemple :

```
i++   ⇔  i=i+1
i--   ⇔  i=i-1
i+=3  ⇔  i=i+3
i*=2  ⇔  i=i*2
```

2.3.4 Entrées sorties à l'écran

On peut afficher du texte et/ou des valeurs de variables à l'écran ainsi que lire des valeurs que l'utilisateur tape au clavier. Pour cela, on fait appel aux fonctions `printf()` et `scanf()` dont les syntaxes générales sont respectivement :

```
printf(format, liste de variables);
scanf(format, liste de &variables);
```

Ces fonctions possèdent 2 arguments :
- La **liste de variables** est facultative. Elle contient un ensemble de variables dont on veut afficher (ou lire) les valeurs. Lorsqu'il y a plusieurs variables, elles sont séparées par des virgules. Lors d'une lecture avec `scanf()`, chaque nom de variable doit être précédé du symbole `&` (voir section 2.3.8 sur les pointeurs).
- Le **format** est une chaîne de caractères qui contient du texte à afficher et la manière d'afficher (ou de lire) les valeurs des variables lorsqu'il y en a. Dans ce cas, à chaque variable doit correspondre une séquence de type `%fmt` où `fmt` indique la manière d'afficher la variable et dépend du type de la variable (entier, réel...). Les différents formats sont présentés dans la table 2.2. **Il extrêmement important**

type	format	format précis
entier par défaut	`%d`	`%5d`, `%05d`
entier court	`%d`	
entier long	`%ld`	
réel simple précision	`%f, %e`	`%10.7f, %10.4e`
réel double précision	`%lf, %le`	
chaîne de caractère	`%s`	`%9s`

TABLE 2.2 – *Syntaxe des formats pour chaque type.*

que le format soit adapté au type de variables à lire ou à afficher. Faute de quoi, les valeurs lues ou affichées sont erronées. En cas de doute, ne pas hésiter à utiliser l'option de compilation `-Wformat` qui vérifie cette compatibilité. Il existe aussi certains caractères spécifiques que l'on peut insérer dans le format comme : `\n` pour nouvelle ligne,`\t` pour une tabulation, `\f` pour une nouvelle page, `\b` pour un backspace, `\r` pour un retour chariot...

Ainsi par exemple :

```
int i=2 ; float x=1.2 ; double y=2.3;
printf("les valeurs sont i = %d \n \t x = %8.5f , y = %lf \n",i,x,y);
```

donne le résultat suivant :

```
les valeurs sont i = 2
        x =  1.200000 , y = 2.3
```

Et on peut lire une valeur comme dans l'exemple suivant :

```
float x ;
printf("Entrez la valeur de x...");
scanf("%f",&x);
```

Application 1a :

- Lire au clavier les valeurs de deux entiers i et j. Afficher leur somme, leur différence, leur produit et leur rapport.

Pour aller plus loin :

- stocker la valeur 1./3 dans une variable `x` de type réel simple précision et dans une variable `y` de type réel double précision. Afficher ces deux variables avec 20 chiffres significatifs.
- Déclarer une variable `int x=1;` et une variable `double y=1./3;`. Afficher `x` avec une format de type reel et `y` avec un format de type entier. Que se passe-t-il ? Inverser l'ordre des `printf`...
- Essayer de compiler avec l'option `-Wformat`
- Déclarer une variable `x` (réel double par exemple), la lire avec un format inadapté (réel simple ou entier par exemple). Afficher le résultat.

2.3.5 Entrées sorties dans des fichiers

L'écriture et la lecture dans un fichier est très similaire aux opérations à l'écran. Elles se font grâce aux fonctions `fprintf()` et `fscanf()`. La syntaxe est très proche et ne présente qu'un seul argument en plus, une variable qui repère le fichier dans lequel lire ou écrire :

```
fprintf(fichier,format,liste de variables);
fscanf(fichier,format,liste de &variables);
```

Cependant, pour pouvoir lire ou écrire dans un fichier, il faut procéder en quatre étapes :

- La déclaration du fichier : `FILE *fichier;`
- L'ouverture du fichier : `fichier=fopen("nom du fichier","action");`
  L'argument `action` désigne le type de l'opération qui sera effectuée sur le fichier :
  - `"r"` : fichier ouvert en lecture uniquement
  - `"w"` : fichier ouvert en écriture uniquement (en écrasant un fichier existant)
  - `"a"` : fichier ouvert en écriture uniquement (en écrivant à la fin d'un fichier existant)
  - `"r+"` ou `"w+"` : fichier ouvert en lecture et écriture

  Si le fichier n'existe pas, il est créé, sinon, l'opération dépend de l'argument `action`.

- Une série d'écritures et/ou de lectures : `fprintf(); fscanf();`
  À chaque appel d'une de ces deux fonctions, l'écriture/lecture reprend là où le dernier appel à une de ces fonctions s'est terminé.

- la fermeture du fichier : `fclose(fichier);`

Ainsi par exemple :

```
FILE *fichier;
fichier = fopen("mon_fichier.txt","w");
fprintf(fichier,"ligne %d \n",1);
fprintf(fichier,"ligne %d \n",2);
fprintf(fichier,"ligne %d \n",3);
fclose(fichier);
```

Application 1b :

– Modifier le programme de l'exercice précédent pour écrire le résultat des opérations demandées (somme, différence, produit, rapport) dans un fichier `resultat.txt`.

2.3.6 Conditions

Comme dans tous les langages, le C possède une structure condition `if/then`. Les deux syntaxes possibles sont les suivantes :

```
if(condition){instructions si vrai;}
if(condition){instructions si vrai;}else{instructions si faux;}
```

Les conditions correspondent à des comparaison qui s'effectuent grâces aux opérateurs suivants :

égal ?	différent ?	supérieur ?	supérieur ou égal ?	inférieur ?	inférieur ou égal ?	et ?	ou ?
`==`	`!=`	`>`	`>=`	`<`	`<=`	`&&`	`||`

Ainsi par exemple :

```
if(n%2==0){printf("le nombre n est   pair");}
else      {printf("le nombre n est impair");}
```

Application 2 :

– écrire un programme qui calcule les racines réelles du polynôme du second degré $ax^2 + bx + c$ (a, b, c étant choisis au clavier par l'utilisateur) lorsqu'elles existent (penser à utiliser le déterminant...).

2.3.7 Boucles

On peut aussi réaliser des boucles dont le but est d'effectuer un grand nombre de fois la même opération.

- Boucle For : `for(i=0; i<n; i=i+1){instructions;}`
- Boucle While : `while(condition){instructions;}`
  Remarque : il existe une syntaxe un peu différente : `do{instructions;}while{condition}`

Par exemple :

```
unsigned int i,n=10;
for(i=0; i<n; i++){printf("l'indice courant est i=%d \n",i);}
```

Application 3 :

– écrire un programme qui remplit un tableau `tableau` de taille `n` donnée par des valeurs saisies au clavier, et qui affiche ensuite les valeurs de ce tableau dans le sens inverse.

Application 4 :

– écrire un programme qui calcule la factorielle d'un nombre `n` donné et qui l'affiche.

2.3.8 Pointeurs

Il s'agit ici de programmation avancée. Mais les pointeurs peuvent être utiles pour comprendre certains aspects de passage d'argument dans les fonctions (voir section suivante).

Les pointeurs sont des variables qui ne contiennent pas de valeurs, mais les adresses mémoire qui contiennent les valeurs d'autres variables. On dit que les pointeurs *pointent* vers des variables. Elles s'utilisent donc de manière légèrement différente des autres variables.

- Déclaration :
  Les déclarations se font de manière similaire à celles des variables standards, mais avec le symbole `*` devant le nom du pointeur : `type *ptr;`. Par exemple : `int *i;` ou `float *x;`.
- Affectation :
  On affecte un pointeur avec le signe `=` comme une variable normale. Par exemple : `ptr1 = ptr2;`. Sachant que l'adresse d'une variable standard s'obtient avec le symbole `&`, on peut faire pointer un pointeur vers l'adresse d'une variable standard avec la syntaxe suivante : `ptr = &var;`.
- Opérations :
  En général, on ne fait pas d'opération sur les adresses mémoires elles-mêmes (c'est à dire sur les pointeurs eux-mêmes). Par contre, on peut manipuler les variables vers lesquels les pointeurs pointent. Ainsi, `*ptr=2.5;` attribue la valeur 2.5 à la variable pointée par le pointeur `ptr`, et `x=*ptr*2;` affecte deux fois la valeur pointée par le pointeur `ptr` à la variable standard `x`.

2.3.9 Fonctions

Les fonctions servent à effectuer un ensemble d'opérations et (le plus souvent) à retourner une valeur résultant d'un calcul. Leur but est de faciliter l'écriture d'un programme compliqué, d'augmenter sa lisibilité et ses possibilités d'évolution.

Définition et appel :

On distingue deux types de fonctions :

- Les fonctions qui calculent et retournent une valeur
  Leur syntaxe est similaire à celle du programme principal, à la différence près qu'il faut définir son type (c'est à dire le type du résultat retourné : entier, réel...), qu'il peut admettre des arguments dont il faut préciser le type, et qu'il retourne un résultat :

```
type nom_de_fonction(type1 arg1, type2 arg2...){
    instructions;
    instructions;
```

```
    return resultat;
}
```

Par exemple, une fonction qui retourne $\cos x + \sin x$ se définira de la manière suivante :
`float sinpluscos(float x){return sin(x)+cos(x);}`.
L'appel à une fonction qui retourne un résultat se fait de la manière suivante :
`var = nom_de_fonction(arg1,arg2...);`.
Ainsi l'appel à la fonction de l'exemple précédent sera fait de la manière suivante : `y=sinpluscos(x);`

- Les fonctions qui ne retournent aucune valeur
  La syntaxe est la même sauf que le type est `void` et qu'il n'y a pas de ligne `return resultat;`. Par exemple :
  `void affiche(float x){printf("la valeur de x est %f",x);}`
  Un appel à une telle fonction se fait sans affectation :
  `nom_de_fonction(arg1,arg2...);`
  Par exemple ici l'appel à la fonction définie précédemment se fait de la manière suivante : `affiche(35.1);`.

Localisation

La définition des fonctions se fait en dehors du programme principal `main`, soit avant, soit après :
- Si la définition est donnée avant le programme principal, alors la fonction est bien connue lors du premier appel à la fonction, et il n'y a rien de plus à faire.
- Si la définition est donnée à la suite du programme principal, alors il faut aussi *déclarer* la fonction avant le programme principal de manière à ce qu'elle soit connue lors du premier appel à cette fonction. La déclaration d'une fonction se fait de la manière suivante :
  `type nom_de_fonction(type1 arg1, type2 arg2...);`

Au bilan voici un petit exemple de programme global incluant les deux types de fonctions :

```
#include<stdio.h>
#include<math.h>
float sincos(float x);
void affiche(float x);
main(){
    float y=3.4;
    affiche(sincos(y));
}
float sincos(float x){return sin(x)+cos(x);}
void affiche(float x){printf("Le resultat est %f \n",x);}
```

Arguments de fonctions

Ce qu'il advient de variables passées en argument lorsqu'une fonction est exécutée et qu'elle modifie leur valeur dépend de la nature des variables.
- Les tableaux sont par défaut toujours passés par **adresse**, même si aucun pointeur n'est utilisé explicitement. Les valeurs des tableaux passés en argument de fonction sont donc **toujours** modifiées lors d'un appel.
- Les scalaires sont quant à eux passés par **valeur** : une copie de la variable est en réalité passée à la fonction, si bien que même si la fonction modifie la valeur de cette copie, la valeur de la variable originale n'est pas affectée. Ainsi le code :

```
void mafonction(double x){x = x+3;}
main(){
   double x=2;
   mafunction(x);
   printf("x = %lf \n",x);
}
```

affichera le résultat `x = 2`. Si l'on désire modifier les valeurs des variables passées en argument, il faut les passer par **adresse**. Ce ne sont donc pas les variables elle-même qui sont passées, mais des pointeurs qui pointent vers les adresses de ces variables. Dans ce cas, une définition (et/ou une déclaration) doit faire apparaître clairement que les arguments passés sont des pointeurs, et les instructions doivent manipuler les valeurs pointées par ces pointeurs. Lors de l'appel à la fonction, il faut également spécifier que l'on passe les adresses avec le symbole `&`. L'exemple précédent doit donc être adapté de la manière suivante :

```
void mafonction(double *x_ptr){*x_ptr = *x_ptr+3;}
main(){
   double x=2;
   mafunction(&x);
   printf("x = %lf \n",x);
}
```

et affiche le résultat `x = 5`.

Application 5 :

- Reprendre le programme de l'exercice précédent et définir une fonction qui calcule la factorielle d'un nombre `n`. L'appeler dans le programme principal et afficher la factorielle de plusieurs nombres.

Application 6 :

- écrire un programme qui calcule le développement en série entière de l'exponentielle $e^{-x} \approx \sum_{k=0}^{n}(-x)^k/k!$ et comparer cette valeur approchée à la valeur obtenue avec la fonction `exp` (le programmeur averti notera qu'une méthode qui n'utilise pas la fonction factorielle permet au programme de s'exécuter beaucoup plus rapidement).
