Introduction à la programmation avec C

1 Éléments de syntaxe

1.1 Organisation d'un fichier C

Voici un exemple de fichier C :

```c
#include <stdio.h>
int main(){
 printf("hello world\n"); //Affiche une phrase
}
```

- Inclusion des biliothèques avec `#include`
- On définit une fonction `int main(int argc, char* argv[])` ou `int main()`
- Tout le code, sauf les définitions de variables globales, est dans le `main`.
- Commentaires avec `//commentaire` ou `/*commentaire*/`
- On termine toutes les instructions avec `;` .

1.2 Compilation

On utilise le compilateur gcc, l'option -o permet de nommer le fichier de sortie :

```
gcc -o sortie.out entree.c
```

Pour tester on exécute le fichier de sortie :

```
./sortie.out
```

1.3 Les types

C est un langage fortement typé mais autorise des changements de types (ou "cast") automatiques.

`sizeof()` pour connaitre la taille mémoire d'un type.

- Entiers : `int`. On ajoute `unsigned` pour n'avoir que des entiers positifs.
- Flottants (approximation des réels) : `float`.
- Caractères : type `char`, par exemple `'a'`. Les caractères sont assimilés à des entiers (stockés sur 1 octet) par la table ASCII.
- Chaines de caractères : `char*`, par exemple `"abc"`.
- Booléens : `true` et `false`.
- C possède un type `void` pour indiquer l'absence de valeur.

1.4 Les opérateurs

On dispose d'opérateurs, comme `+`, `-`, `*`, `/` qui calculent sur les entiers et les flottants et `%` pour le modulo. On précise qu'appliquer un opérateur à deux entiers renvoie un entier, l'appliquer à deux flottants renvoie un flottant et l'appliquer à un entier et un flottant renvoie un flottant.

On dispose également des opérateurs logiques "et" : `||`, "ou" : `&&` et "non" : `!` et des opérateurs de comparaison `==` (égalité), `<`, `>`, `<=`, `>=` et `!=` (différence).

1.5 Définir une variable

La syntaxe est la suivante : `type nom_variable = valeur;`

Exemple 1.1.

```c
unsigned int x = 3;
float y = 4.5;
```

La portée d'une variable est limitée au code en-dessous de sa définition et entre les accolades { et } les plus proches. Si la variable est définie au même niveau que `main`, en dehors de toute fonction, alors elle est globale. **Une variable globale existe partout dans le code, une variable locale n'existe qu'entre les accolades les plus proches d'elle**

En C, les variables sont mutables (modifiables) sauf si elles sont définies avec le mot-clé `const`.

Exemple 1.2.

```
const int VARIABLE = 3;
```

Remarque 1.1. Pour différencicer les constantes et les variables normales, on écrit souvent le nom des constantes en majuscules.

On peut changer la valeur d'une variable grâce aux opérateurs d'assignation : `=, +=, *=, -=, /=`.

Exemple 1.3.

```
float y = 4; //y vaut désormais 4.0
y += 0.1; //y vaut désormais 4.6
```

1.6 Les librairies utiles

On importera toujours :

- `<assert.h>` : contient la commande `assert` pour vérifier une condition et lever une erreur.
- `<stdbool.h>` : définit les booléens.
- `<stddef.h>` : définit la constante `NULL` et le type `size_t` (le type de la taille en mémoire).
- `<stdint.h>` : utilitaires pour les entiers.
- `<stdio.h>` : fonctions d'entrée/sortie et affichage.
- `<stdlib.h>` : fonctions système fondamentales : `malloc` par exemple.

On peut également inclure du code qu'on a écrit nous-même dans un autre fichier grâce à la commande `#include`.

1.7 Les structures de contrôle

Les blocs, entre { } permettent de séparer des séries d'instructions différentes.

Conditionnelle

```
if (condition) {...}
else {...}
```

`condition` est une expression de type `bool`. Le premier bloc contient ce qui est effectué lorsque la condition est vraie. Le bloc après `else` contient les instructions à effectuer si la condition est fausse.

On a quelques variantes :

```
if (condition) {...}
```

sans le else, ou pour un instruction unique on peut retirer les accolades.

```
if (condition) instruction;
else {...}
```

On peut également écrire des expressions conditionnelles avec `int a = condition ? val1 : val2`.

Exemple 1.4.

```
int x = 3;
int y = x>3 ? 3 : 1;
```

La variable `y` vaut 3 si `x>3` et 1 sinon.

Boucles

La boucle tant que : `while (condition) {...}`

```
int i=0;
while (i<n){
  ...
  i = i +1 ;
}
```

La boucle pour : `for (int i=0; i<n ; i=i+1) { ... }`

Remarque 1.2. On peut changer les 3 conditions à notre guise. Cette boucle n'est rien d'autre que :

```
int i=0;
while (i<n){
  ...
  i+=1;
}
```

1.8 Les fonctions

Une fonction est définie avec un type de retour et un type d'entrée pour chacun des arguments.

L'instruction **return** est obligatoire sauf si le type de retour est **void**. L'instruction **return** signale la fin de la fonction. On peut utiliser **return** sans argument pour terminer une fonction dont le type de retour est **void**.

Exemple 1.5.

```
int quotient_euclide(int a, int b){
  int q = 0;
  while(a>=b){
    a-=b;
    q+=1;
  }
  return q;
}
```

Les fonctions peuvent être récursives. Aucune syntaxe particulière n'est nécessaire, il suffit d'appeler le nom de la fonction dans le corps de la fonction.

Exemple 1.6. Exemple de fonctions mutuellement récursives :

```
bool even(unsigned int x);
bool odd(unsigned int x) { return x !=0 && even (x-1);}
bool even(unsigned int x) { return x ==0 || odd (x-1);}
```

2 Les pointeurs et les structures

2.1 Pointeurs

Un pointeur **p** est une variable dont la valeur est une adresse en mémoire. La case mémoire à cette adresse contient une valeur d'un certain type.

Si le type de la valeur est un type τ quelconque, alors **p** a pour type $\tau*$.

- Déréférencement d'un pointeur : on accède à la valeur de la case pointée par **p** en écrivant `*p`. On peut la modifier, par exemple `*p += 1` ajoute 1 à la valeur de la case pointée par **p**.
- Création d'un pointeur vers une nouvelle case : `type *p = malloc(sizeof(type))` (remplacer type par le type)
- Création d'un pointeur vers la case d'un variable **x** qui existe déjà : `type *p = &x` (remplacer type par le type)

Remarque : on peut (et on doit) supprimer un pointeur créé avec **malloc** en utilisant `free`.

Tous les pointeurs peuvent prendre la valeur spéciale **NULL**. **NULL** est l'unique pointeur de type **void*** mais il peut être casté implicitement en tous les types.

Un pointeur qui vaut NULL ne peut pas être déréférencé.

2.2 Tableaux

2.2.1 Création

Définition 2.1.

- Tableau vide statique : taille connue à la compilation

  ```
  int tab[4];
  ```

  crée un tableau d'entiers de taille 4.

- Tableau rempli statique

```
int tab[4] = {0,1,2,3};
```

ici on précise la taille. On peut aussi ne pas la préciser :

```
char tab[] = {'a', 'b', 'c', 'd', 'e'};
```

ou initialiser toutes les cases du tableau à 0 :

```
int tab[4] = {0};
```

- Tableau vide dynamique : taille connue à l'exécution

```
int* tab = malloc(4*sizeof(int));
```

On peut (et on doit) supprimer un tableau dynamique de la mémoire avec `free`.

2.2.2 Utilisation

Pour accéder à l'élément i : `tab[i]`.

On peut modifier chque case comme une variable : `tab[i] = 6;`

La taille du tableau ne peut pas être retrouvée avec juste le tableau, il faut toujours donner la taille d'un tableau quand on donne un tableau (en entrée d'une fonction par exemple)

2.3 Tableaux multidimensionnels

Première manière : un tableau de pointeurs vers des tableaux.

```
int **mat = malloc(n*sizeof(int*));
for (int i=0; i<n; i+=1){
  mat[i] = malloc(m*sizeof(m));
}
```

Pour accéder à l'élément (i, j) : `mat[i][j]`. Cette instruction suit les pointeurs pour trouver la case.

Deuxième manière : un tableau statique de taille $m * n$, "mis à plat".

```
int mat2[12][36];
```

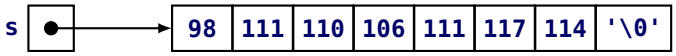
On accède toujours à l'élément (i, j) avec `mat[i][j]`. Cette instruction cherche l'élément $n * i + j$ du tableau.

Pour ce 2ème type il faut toujours le passer en argument comme étant `int mat[n][m]`.

2.4 Chaînes de caractères

Une chaîne de caractères (**`char*`**) est un tableau de **`char`**. Pour indiquer la fin de la chaîne, on utilise une sentinelle : le caractère **`'\0'`**.

Par exemple la chaîne `char* s = "bonjour"` donne en mémoire :



Ce caractère final permet notamment de calculer la longueur de la chaîne. Ce qui n'est pas prévu avec un tableau normal.

Définition 2.2. Il y a des fonctions préécrites :

- **`strlen(s)`** donne la longueur d'une chaîne de caractères **`s`**.
- **`strcmp(s1,s2)`** compare deux chaînes pour l'ordre lexicographique. Le résultat est un entier signé, zéro s'il y a égalité.ères des deux chaînes.
- **`atoi(s)`** transforme une chaîne représentant un entier en un entier. La fonction produit une erreur si **`s`** contient autre chose que des chiffres.

Remarque 2.1. Les chaînes définies entre guillemets sont préallouées et immuables. Les autres chaînes se modifient comme un tableau.

2.5 Types structurés

- Définition d'un type structuré avec :

```
struct complexe {double re; double im;};
```

  Cette définition définit le type `struct complexe`.

- Définition d'une instance de la structure :
  - avec définition des champs **re** et **im** :

```
int main() {
  struct complexe z = { .re = 0.1, .im=0.3};
  }
```

  - sans définition des champs **re** et **im** :

```
int main() {
  struct complexe z;
  }
```

- Modification des champs :

```
int main() {
  struct complexe z = { .re = 0.1, .im=0.3};
  z.re = 2.;
  }
```

- On dispose d'une manière alternative de créer un structure, avec un pointeur.

```
int main() {
  struct complexe *z = malloc(sizeof(struct complexe));
  z -> re=0.1;
  z -> im=0.2;
  }
```

- Passage en argument d'une structure par valeur : lorsqu'on passe une structure en argument d'une fonction, elle est intégralement copiée. Pour éviter cela, on peut passer en argument un pointeur plutôt.

```
void incr_re (struct complexe x){
  x.re = x.re +1;
  }
int main () {
  struct complexe z = {.re=0.1, .im=3.};
  incr_re (z);
  printf("%f",z.re)
  }
```

- On peut renommer le type `struct nom_a` en un nom plus court `nom_b` avec la syntaxe

```
typedef struct nom_a nom_b;
```