10 Consolidation des bases

Les vacances sont passées, mais nous revoilà à étudier le langage C ! On peut faire quelques rappels rapides avant d'avancer.

10.1 Remise dans le bain rapide

10.1.1 Un programme en C

Dans un programme en C :

1. Vous commencez par importer les bibliothèques dont les fonctionnalités vous seront utiles avec `#include`.
2. Vous listez les déclarations des fonctions que vous pourriez vouloir définir.
3. Vous écrivez les instructions à suivre au lancement de votre application dans la fonction `main`.
4. Vous listez les déclarations de vos fonctions.

Rappelez vous, ça ressemble à ça :

```c
#include <stdio.h>
#include <stdlib.h>
/* ... */

/* Déclaration des fonctions */

int main() {

  /* Instructions de l'application */

  exit(EXIT_SUCCESS);
}

/* Définition des fonctions */
```

Pour compiler un tel programme depuis un fichier source `main.c`. On peut utiliser `gcc` et préciser le nom du programme avec l'option `-o` :

```
gcc -o monProgramme main.c

./monProgramme
```

10.1.2 Variables

En langage C, les variables sont typées. Les types atomiques sont :

Entiers :

```
char  /* 1 octet */
short /* 2 octets */
int   /* 4 octets */
long  /* 8 octets */
unsigned type /* positif */
```

Flottants :

```
float       /* 4 octets */
double      /* 8 octets */
long double /* 16 octets */
```

Vous devez les utiliser lorsque vous déclarez une variable ou pour forcer un changement de type :

```
unsigned int valeurPositive = 42; /* Affectation de 42 */
valeurPositive = 0x2a; /* Réaffectation en hexadécimal */
```

Pour imprimer des caractères dans le terminal et afficher les valeurs de vos variables vous avez la fonction d'affichage formaté `printf` :

```
printf("caractere : \'%c\'\n", '@');
printf("entier : %d\n", 42);
printf("hexadécimal : %x\n", 0x2a);
printf("entier positif : %u\n", 3000000000u);
printf("entier long : %ld\n", 42000000000);
printf("flottant : %g\n", 3.14);
printf("adresse : %p\n", NULL);
```

Pour lire les caractères saisis par l'utilisateur dans son terminal, vous avez `scanf`. Notez que vous devrez passer une adresse à `scanf` pour réussir votre affectation :

```c
int entier;
scanf("%d", &entier);
float flottant;
scanf("%f", &flottant);
char caractere;
scanf(" %c", &caractere);
```

10.1.3 Opérations

Souvent il va être intéressant de combiner les valeurs en utilisant des opérateurs. Les opérateurs classiques sont les suivants :

```c
/*first et second deux variables de type entier ou flottant*/
first + second /* addition */
first - second /* soustraction */
first * second /* multiplication */
first / second /* division */
first % second /* modulo : entiers */
```

Notez que dans le cas d'entiers, vous seriez amenés à utiliser une coercition pour gérer un dépassement de capacité d'un `int` ou pour réaliser une division avec résultat flottant :

```c
int first, second;
/* ... */
long multiplication = (long)first * second;
float division = (float)first / second;
```

En langage C, il existe des raccourcis pour incrémenter une valeur ou utiliser un opérateur :

```c
int valeur = 1;
valeur = valeur + 1; /* ajoute 1 */
valeur += 1;         /* ajoute 1 */
valeur++;            /* ajoute 1 */
++valeur;            /* ajoute 1 */
```

10.1.4 Tests et disjonctions

Il est possible de réaliser une disjonction de cas sous condition à l'aide d'un `if-else` :

```
int first, second;
scanf("%d %d", &first, &second);
if(first > second) {
  printf("%d est plus grand.\n", first);
} else if(first < second) {
  printf("%d est plus grand.\n", second);
} else {
  printf("les deux sont égaux.\n");
}
```

On différencie le test d'égalité `==` de l'affectation `=`. Les opérateurs de comparaison sont les suivants :

```
/* égalité : */
a == b
/* plus petit strict : */
a < b
/* plus petit ou égal : */
a <= b
```

```
/* différence : */
a != b
/* plus grand strict : */
a > b
/* plus grand ou égal : */
a >= b
```

Pour assembler les valeurs de vérité renvoyées par ces opérateurs, on utilise les opérateurs suivants :

```
a && b /* vrai lorsque les deux le sont */
a || b /* vrai lorsque d'un l'est */
! a /* vrai lorsque faux et réciproquement */
```

Il est possible de synthétiser une affectation dans un `if-else` à l'aide d'une opération ternaire :

```c
if(a < b) {
  res = a;
} else {
  res = b;
}
```

⇔

```c
res = (a < b) ? a : b;
```

Pour des conditions simples d'égalité à une constante, il est possible d'utiliser un `switch` pour se brancher à un bloc correspondant :

```c
switch(expression) {
        case 1 : {
                /* instructions */
        } break;
        case 2 : {
                /* instructions */
        } break;
        default : {
                /* instructions */
        }
}
```

10.1.5 Boucles

Il est possible d'automatiser la répétition d'instructions à l'aide de boucles.
On utilise :

- `while` pour répéter les instructions tant qu'une condition est vraie.

```c
while(condition) {
        /* instructions */
}
```

- `do-while` pour répéter à nouveau les instructions lorsqu'une condition est vérifiée.

```
do {
        /* instructions */
} while(condition);
```

- `for` lorsque la boucle while peut s'écrire avec une initialisation et une évolution des données utilisées pour la condition.

```
for(initialisation; condition; evolution) {
        /* instructions */
}
```

Il est aussi possible de relancer la boucle prématurément à l'aide du mot-clé `continue` ou de la stopper avant vérification de la condition par le mot-clé `break`.

10.1.6 Fonctions

Une fonction se définit par un type de retour, un nom d'appel et des paramètres :

```
typeRetour nomFonction(/* paramètres */) {
        /* instructions */
}

/* Exemple de fonction d'addition d'entiers : */
int addition(int first, int second) {
        int res; /* variable locale à la fonction */
        res = first + second;
        return res; /* renvoi lors de l'appel */
}
```

Cette fonction peut être déclarée en amont par sa signature (type de retour, nom et type des paramètres) et appelée par son nom :

```
/* Exemple de fonction d'addition d'entiers : */
/* déclaration */
int addition(int, int);
```

```c
int main() {
        /* appel */
        int deux = addition(1, 1);
        exit(EXIT_SUCCESS);
}

/* définition */
int addition(int first, int second) {
        return first + second;
}
```

10.1.7 Tableaux

Un tableau est l'outil en langage C qui permet de créer une liste de taille donnée pour une type de variable que l'on souhaite répéter :

```c
type tableau[TAILLE_CONSTANTE];
type tableau[] = {valeur1, valeur2, ..., valeurN};

int liste[] = {1, 2, 3};
liste[1]; /* accès au second élément : d'indice 1 */
```

Les tableaux sont utilisés pour gérer les chaînes de caractères (se terminant par un marque de fin `'\0'`) :

```c
char chaine[] = "Hello ESGI !";
printf("%s\n", chaine); /* affichage de chaine */
scanf("%s", chaine); /* lecture d'un mot au clavier */
```

Il est possible de gérer des tableaux à plusieurs dimensions :

```c
/* tableau à deux dimensions */
int grille[HAUTEUR][LARGEUR] = {
  {0, 0, 0, 0},
  {0, 1, 2, 0},
  {0, 0, 0, 0}
```

```
};

grille[ligne][colonne]; /* accès au tableau */

/* passage d'un tableau à deux dimensions */
void afficherGrille(int largeur, int hauteur,
    grille[hauteur][largeur]) {
  /* instructions */
}
```

10.1.8 Pointeurs

Toute donnée en mémoire possède une adresse, les pointeurs sont l'outil qui permet d'utiliser ces adresses dans votre programme. Nous avons vu que nous pouvons récupérer l'adresse d'une variable à l'aide de l'opération unaire `&`. Pour sauvegarder cette adresse dans une variable, on construira une pointeur sur cette variable dont le type prendra une étoile par rapport à celui de la variable. L'accès à la donnée pointée se fera ensuite à l'aide d'un déréférencement par l'opérateur unaire `*` :

```
int variable = 42;
int * pointeur = &variable;
*pointeur = 1337;
printf("%d\n", variable); /* affiche 1337 */
```

Pour rappel, un tableau correspond à l'adresse de son premier élément (d'indice 0), les autres étant alignés en mémoire après celui-ci. La manipulation d'un tableau peut aussi s'appliquer avec un pointeur :

```
int tableau[] = {1, 2, 3, -1};
int * pointeur = tableau;
int i;
for(i = 0; pointeur[i] >= 0; ++i) {
  printf("%d\n", pointeur[i]);
}
```

À noter qu'un pointeur peut changer de valeur (pointer vers une autre adresse), ce qui permet par exemple de jouer avec l'arithmétique des pointeurs :

```c
char texte[] = "Hello !";
char * pointeur = NULL;
for(pointeur = texte; *pointeur != '\0'; ++pointeur) {
  if(*pointeur >= 'a' && *pointeur <= 'z')
    *pointeur += 'A' - 'a';
}
printf("%s\n", texte); /* affiche "HELLO !" */
```

L'un des grands intérêts des pointeurs est de pouvoir manipuler des plages mémoire allouées dynamiquement : de taille décidée à l'exécution du programme.

```c
float * notes = NULL;
float somme = 0;
int nombre;
int i;
printf("Combien de CC ? ");
scanf("%d", &nombre);
if(nombre <= 0) {
  printf("Pas de notes pas de moyenne.\n");
  exit(EXIT_FAILURE);
}
/* allocation dynamique depuis le nombre donné par l'utilisateur
↪  */
if((notes = (float *)malloc(sizeof(float) * nombre)) == NULL) {
  printf("Erreur d'allocation.\n");
  exit(EXIT_FAILURE);
}
for(i = 0; i < nombre; ++i) {
  scanf("%f", notes + i);
  somme += notes[i];
}
printf("La moyenne de ");
for(i = 0; i < nombre; ++i) {
  if(i && i == nombre - 1) printf(" et ");
  else if(i > 0) printf(", ");
```

```
  printf("%g", notes[i]);
}
printf(" est %g\n", somme / nombre);

free(notes);
notes = NULL;
exit(EXIT_SUCCESS);
```

Les principales fonctions d'allocation sont les suivantes :

```
/* allouer une plage mémoire */
malloc(/*taille mémoire en octets*/)

/* allouer un tableau avec chaque élément à 0 */
calloc(/*taille tableau*/, /*taille élément*/)

/* modifier la taille d'une plage allouée */
realloc(/*plage à modifier*/, /*nouvelle taille en octets*/)
```

À jour et prêt pour la suite ? Pas de panique, on fait des exercices là dessus pour se remettre dans le bain.

10.2 Entraînement

Exercice noté 37 (⋆⋆⋆ Statistiques sur liste d'entiers).

Écrire un programme qui lit une liste d'entiers et affiche les statistiques suivantes :

- Valeur minimale.
- Valeur maximale.
- Moyenne des valeurs.

```
Entrez des entiers positifs : 14 12 10 8 7 -1
min : 7
max : 14
moyenne : 10.2
```

Exercice noté 38 (⋆⋆⋆ Extraire information d'une chaîne de caractères).

Compléter le code suivant de manière à extraire les informations de la chaîne de caractère. Puis afficher ces informations :

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

int main() {
  const char * infos = "Linus Torvalds 52 ans C";
  /* TODO : extraire les informations */
  exit(EXIT_SUCCESS);
}
```

```
Prenom : Linus
Nom : Torvalds
Age : 52
Parle couramment la langue C
```

Exercice noté 39 (⋆⋆⋆ Mini-interpréteur sur liste d'entiers).

Un ami a entendu parler du langage Brainfuck. C'est un langage où vous pouvez vous déplacer dans la mémoire et y changer des valeurs. Il aimerait que vous en codiez un mini-interpréteur dans une plage de taille donnée par l'utilisateur en suivant les règles suivantes :

- `'+'` ajoute 1 à la case mémoire regardée par le curseur.
- `'-'` retire 1 à la case mémoire regardée par le curseur.
- `'='` égalise toutes les cases mémoire à la valeur de la case mémoire regardée.
- `'>'` déplace le curseur vers la droite (retour au début si dépasse de la plage mémoire).
- `'<'` déplace le curseur vers la gauche (retour à la fin si dépassé de la plage mémoire).
- `'.'` affiche le contenu de la plage mémoire.

```
taille de la mémoire : 4
.
[0, 0, 0, 0]
+.
[1, 0, 0, 0]
=.
[1, 1, 1, 1]
>++.
[1, 3, 1, 1]
=.
[3, 3, 3, 3]
<--.
[1, 3, 3, 3]
<++++.
[1, 3, 3, 7]
-->>-.
[1, 2, 3, 5]
```

Exercice noté 40 (⋆⋆⋆ Jeu du Morpion).

Coder sous console un jeu de Morpion :

- Jeu à deux joueurs dans une grille de 3×3.
- Tour par tour chacun place un pion.
- Lorsque 3 pions d'un même joueur sont alignés, il gagne.

```
+-+-+-+
|X| |O|   Le joueur X gagne !
+-+-+-+
| |X| |
+-+-+-+
|O| |X|
+-+-+-+
```

Exercice noté 41 (⋆⋆⋆⋆ Enregistrement et recherche de numéros).

Écrire un programme qui demande une liste de noms et associe à chaque nom un numéro. Une fois que l'utilisateur a validé sa liste, lui proposer de rechercher un nom. Si le nom a été renseigné, on lui affiche le numéro associé. Une sortie de ce programme pourrait ressembler à la suivante :

```
Nom (None pour arrêter) : Personne
Numéro : 42
Nom (None pour arrêter) : Moi
Numéro : 1337
Nom (None pour arrêter) : Lui
Numéro : 1234
Nom (None pour arrêter) : None
Nom à rechercher (None pour arrêter) :
>>> Personne
Le numéro de "Personne" est 42
Nom à rechercher (None pour arrêter) :
>>> Toi
"Toi" non trouvé.
Nom à rechercher (None pour arrêter) :
>>> Moi
Le numéro de "Moi" est 1337
Nom à rechercher (None pour arrêter) :
>>> None
```