Un tour en C

Plan

1. Bonjour, monde !
2. Compter : Un peu de syntaxe
3. Lire
4. Adresses
5. Tableaux
6. Structures

1. Bonjour, monde !

```
#include <stdio.h>

main()
{
  printf("Bonjour, monde !\n");
}
```

- Un *programme* est une suite de déclarations de variables et définitions de fonctions.
- Tout programme contient une fonction de nom imposé `main` qui est exécutée au lancement du programme.
- Une fonction contient une suite de déclarations de variables locales, suivie d'une suite d'instructions (donc pas de définitions de fonctions impbriquées)

2. Compter : Un peu de syntaxe

```
/*** compter.c ***/
#include <stdio.h>
#define BLANC ' '
#define POINT '.'

main()
{
  int blancs = 0, autres =0;
  char c;

  c = getchar();
  while (c != POINT) {
    if (c == BLANC)
      blancs++;
    else
      autres++;
    c = getchar();
  }
  printf("Le texte comporte\n");
  printf(" %d blancs\n%d autres caracteres\n", blancs, autres);
}
```

Voici un exemple d'emploi:

```
Le texte compte
encore et encore.

Le texte comporte
 4 blancs
28 autres caracteres
```

Quelques explications:

- Les *noms de variables* sont des suites de lettres et chiffres commençant par une lettre. Le caractère de soulignement `_` est une “lettre” autorisée. Les majuscules et minuscules sont distinctes.
- Le *point virgule* “;” fait partie de certaines instructions, et n’est pas un simple séparateur. Doivent être terminées par un point virgule
  - une déclaration, comme
    ```
    char c;
    ```
  - une instruction d’affectation, comme
    ```
    c = getchar();
    autres++;
    ```
  - un appel de fonction, comme
    ```
    printf("Le texte comporte\n");
    ```
  - des instructions de rupture de séquence, comme
    ```
    return;
    break;
    continue;
    ```

  L’instruction conditionnelle `if` ou `while` ou `for` n’ont pas de point virgule final, mais les instructions contenues peuvent en avoir besoin.
- Le *groupement d’instructions* se fait par une paire d’accolades “`{}`”.
- Les *types prédéfinis* en C sont
  ```
  char int float double void
  ```
  Les mots-clés `short` et `long` permettent d’influer sur la taille des types de base. On peut spécifier qu’un entier ou caractère est sans signe en le préfixant de `unsigned`. Un `char` est en fait un entier qui est le code ASCII du caractère.
- La fonction `getchar` retourne le caractère suivant de l’entrée standard.
- La fonction `printf` a un nombre variable d’arguments: le premier est une chaîne de caractères qui contient des signe de formatage (débutant par `%`), les autres sont des expressions. A l’exécution, chaque signe de formatage est remplacé par un des arguments, dans l’ordre d’apparition, pour former le texte à imprimer.

D'autres explications:

Toute expression a une valeur. Dans le cas d'une expression d'affectation, la valeur est la valeur affectée. Ainsi, la valeur de

```
i = 5
```

est la valeur de `i`, donc `5`. On peut affecter cette valeur, ou l'utiliser autrement:

```
j = i = 5;

if ((i = 5) > k)
{...}
```

Ceci simplifie considérablement l'écriture de structures de contrôle. Ainsi, la suite

```
c = getchar();
while (c != POINT) {
  if (c == BLANC)
    blancs++;
  else
    autres++;
  c = getchar();
}
```

est remplacée par:

```
while ( (c = getchar()) != POINT) {
  if (c == BLANC)
    blancs++;
  else
    autres++;
}
```

Variation

Le programme qui suit compte divers paramètres d'un texte:

```
/*** decompte.c ***/
#include <stdio.h>
#define BLANC   ' '
#define VIRGULE ','
#define POINT   '.'
#define RC     '\n'

main()
{
  int nb = 0, nv = 0, np = 0, nl = 0, nc =0;
  char c;

  while ((c = getchar())!= EOF) {
    nc++;
    switch (c) {
      case BLANC : nb++; break;
      case VIRGULE : nv++; break;
      case POINT : np++; break;
      case RC: nl++;
    }
  }
  printf("nombre de caracteres = %d\n", nc);
  printf("nombre de blancs = %d\n", nb);
  printf("nombre de virgules = %d\n", nv);
  printf("nombre de points = %d\n", np);
  printf("nombre de lignes = %d\n", nl);
}
```

Si le programme se lit lui-même, on obtient:

```
nombre de caracteres = 568
nombre de blancs = 114
nombre de virgules = 10
nombre de points = 3
nombre de lignes = 27
```

En fin de lecture, la fonction `getchar` retourne un entier particulier de nom `EOF` (défini à `-1` dans les fichiers d'en-tête).

L'instruction `switch` est un *aiguillage*. Le paramètre détermine quel cas est à considérer, mais

- lorsqu'un cas est traité, l'exécution *continue par défaut avec le cas suivant*; il faut donc sortir de l'aiguillage par `break`
- si aucun cas ne s'applique, on passe en séquence sans erreur. On peut utiliser un champ `default :` pour traiter les cas non considérés.

3. Lire

```
#include <stdio.h>

main()
{
  char nom[20];
  printf("Donner votre nom : ");
  scanf("%s", nom);
  printf("Bonjour, %s\n", nom);
}
```

4. Adresses

Toute donnée est rangée en mémoire à une *adresse*. Un *pointeur* est une variable qui a pour valeur une *adresse*.

L'adresse d'une variable `x` est obtenue par `&x`.

Le contenu se trouvant à l'adresse `p` est obtenu par `*p`

```
/** adresses.c */
int a[2]; /* tableau de deux entiers */
int x=23;
main()
{
   a[0]=123; a[1]=345;
   printf("Valeur  de x: %3d\n",x);
   printf("Adresse de x: %ld\n",&x);
   printf("Valeur  de x: %3d\n",*&x);
   printf("Valeur de a[0]: %3d\n",a[0]);
   printf("Adresse de a[0]: %ld\n",&a[0]);
   printf("Adresse de a[0]: %ld\n",a);
   printf("Valeur de a[0]: %3d\n",*a);
   printf("Adresse de a[1]: %ld\n",&a[1]);
   printf("Adresse de a[1]: %ld\n",a+1);
}
```

```
Valeur  de x:  23
Adresse de x: 4535590
Valeur  de x:  23
Valeur de a[0]: 123
Adresse de a[0]: 4535592
Adresse de a[0]: 4535592
Valeur de a[0]: 123
Adresse de a[1]: 4535594
Adresse de a[1]: 4535594
```

Echange

La fonction `echange` échange la valeur ses arguments

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

echange(int* a, int* b) {
  int x;
  printf("Adresse/valeur du premier arg. : %ld %d\n", a, *a);
  printf("Adresse/valeur du deuxieme arg.: %ld %d\n", b, *b);
  x = *a;
  *a = *b;
  *b = x;
}

main()
{
  int p = 5, q = 7;
  printf("Adresse/valeur de p : %ld %d\n", &p, p);
  printf("Adresse/valeur de q : %ld %d\n", &q, q);
  echange (&p, &q);
  printf("Adresse/valeur de p : %ld %d\n", &p, p);
  printf("Adresse/valeur de q : %ld %d\n", &q, q);
}
```

```
Adresse/valeur de p : 25725260 5
Adresse/valeur de q : 25725258 7
Adresse/valeur du premier arg. : 25725260 5
Adresse/valeur du deuxieme arg.: 25725258 7
Adresse/valeur de p : 25725260 7
Adresse/valeur de q : 25725258 5
```

5. Tableaux

Les indices d'un tableau commencent à 0. La déclaration d'un tableau de `N` entiers se fait par

```
int vec[N];
```

Les éléments du tableau sont `vec[0]`, `vec[1]`, `...vec[N-1]`. La boucle pour a la structure générale

`for` (*initialisation*; *condition de continuation*; *modification*)

où

- l'initialisation est effectuée avant l'entrée dans la boucle;
- le test de la condition de continuation est exécuté *avant* chaque tour dans la boucle;
- la modification est réalisée *après* chaque tour de boucle, juste avant de poursuivre par un nouveau test.

```
for (i=0; i<5; i++)
printf("%d ", i);
```

affiche

```
0 1 2 3 4
```

En particulier

- Si la condition de continuation est fausse dès le départ, la boucle n'est pas exécutée.
- L'un ou plusieurs des arguments peuvent manquer. Ainsi,

  ```
  for (;;)
  ```

  est une boucle sans arrêt.

```
/*** minimax.c ***/

#include <stdio.h>
#define N 20
main()
{
  int min = 0, max = 0, i, n;
  int vec[N];

  scanf("%d", &n);
  for (i=0; i<n; i++)
    scanf("%d", &vec[i]);
  if (n>0)
    min = max = vec[0];
  for (i=1; i<n; i++) {
    if (vec[i] > max)
      max = vec[i];
    if (vec[i] < min)
      min = vec[i];
  }
  printf("Le maximum des %d nombres est %d\n", n, max);
  printf("et leur minimum est %d\n", min);
}
```

Ceci donne par exemple:

```
12
6 -7 41 12 45 56 -34 41 23 65 22 1

Le maximum des 12 nombres est 65
et leur minimum est -34
```

Observons

- L'usage d'un `#define` pour fixer une constante. Le préprocesseur remplace toute occurrence de `N` par `20` avant de compiler; le compilateur trouve donc:

  ```
  int vec[20];
  ```

- `vec` désigne l'adresse du premier élément du tableau. Donc `&vec[i]` est synonyme de `vec+i`. Nous préférons la première écriture !

```
/*** eqsnd.c : equation du second degre ***/
#include <stdio.h>
#include <math.h>  /* pour sqrt */

#define sqr(x)  ((x)*(x))
#define abs(x)  ((x) >= 0 ? x : -(x))

main()
{
  float a, b, c, discr, re, im;

  scanf("%g%g%g", &a, &b, &c);
  if ((a == 0) && (b == 0)) {
    printf("Donnees sans interet\n");
    return;
  }
  if (a == 0) {
    printf("Racine simple reelle : %g\n", -c/b);
    return;
  }
  re = -b/(2*a);
  discr = sqr(b) -4*a*c;
  im = sqrt(abs(discr))/(2*a);

  if (discr >= 0)
    printf("Deux racines relles : %g et %g\n", re+im, re-im);
  else {
    printf("Deux racines complexes : %g +i %g\n", re, im);
    printf("                    et   %g -i %g\n", re, im);
  }
}
```

Par exemple, on obtient:

```
1 1 1
Deux racines complexes : -0.5 +i 0.866025
                    et   -0.5 -i 0.866025

1 5 6
Deux racines relles : -2 et -3

2 3 0
Deux racines relles : 0 et -1.5

0 10 2
Racine simple reelle : -0.2
```

Les *macros* introduites par `#define` peuvent avoir des paramètres; la substitution se fait alors en remplaçant les paramètres. Ici, les deux lignes

```
    discr = sqr(b) -4*a*c;
    im = sqrt(abs(discr))/(2*a);
```

sont développées en

```
    discr = ((b)*(b)) -4*a*c;
    im = sqrt(((discr) >= 0 ? discr : -(discr)))/(2*a);
```

Une *expression conditionnelle* a la forme générale

(*test*) ? *alternant-oui* : *alternant-non*

et a pour valeur l'alternant qui s'applique en fonction du test.
Par exemple, que vaut:

```
    1+ (x == 1) ?  0 :  (1+x)/(1-x)
```

6. Structures

Une *structure* est un groupement d'objets. Par exemple,

```
struct mot {
  int longueur;
  char chaine[MAX];
};
```

définit une structure qui est le regroupement d'un entier et d'un vecteur de `MAX` caractères. Chaque objet est un *champs*, identifié par son *nom* (ici `longueur` et `chaine`.

Le nouveau type est `struct mot` (et non `mot`). On définit un variable de ce type par

```
struct mot w;
```

On accède aux champs par

nom.*champ*

donc ici par

```
w.longueur
w.chaine
```

Si p est un pointeur sur une structure (par exemple

```
struct mot* p;
```

on accède aux champs indifféremment par `(*p)`.*nom* ou, plus commodément, par

`p ->` *champ*

En d'autres termes, "`->`" est une abbréviation de "`(* ).`". Dans notre exemple, le 4ème caractère du mot pointé par `p` est

```
p -> chaine[3]
```

Entiers de Gauss

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

struct gauss {
  int x;
  int y;
};

struct gauss a={1,0},b={0,-123};

void imprimer(struct gauss z) {
  if (z.y > 0)
    printf("%3d + %di\n", z.x, z.y);
  else if (z.y < 0)
    printf("%3d - %di\n", z.x, -z.y);
  else
    printf("%3d\n", z.x);
}

struct gauss plus (struct gauss u, struct gauss v) {
  struct gauss  z;
  z.x = u.x+v.x; z.y = u.y+v.y;
  return z;
}

void pplus (struct gauss u, struct gauss v, struct gauss *z) {
  z->x = u.x+v.x;
  z->y = u.y+v.y;
}

main() {
  struct gauss t, w;
  imprimer(a); imprimer(b); imprimer(plus(a,b));
  pplus(a,b,&t); imprimer(t);
  w = t; imprimer(w);
}
```

On obtient

```
1
0 - 123i
1 - 123i
1 - 123i
1 - 123i
```

Observons

- Toute invocation de `gauss` est précédée de `struct`.
- On peut initialiser une structure comme un tableau, par énumération de ses éléments, si elle s'y prête.
- Une structure peut être le résultat d'une fonction.
- La copie de structures (`w = t;`) fait une copie champ par champ.

Synonymes `typedef`

Le constructeur `typedef` permet de définir des *synonymes* ou *alias* de types. Par exemple

```
typedef struct gauss Gauss;
```

définit `Gauss` comme synonyme de `struct gauss`. Ceci rend équivalentes des écritures comme:

```
void pplus (struct gauss u, struct gauss v, struct gauss *z) {}
void pplus (Gauss u, Gauss v, Gauss *z) {}
```

Plus généralement, on remplace une déclaration de variable par une déclaration de `typedef`. Par exemple:

```
float a[4];
```

définit un vecteur réel `a` de dimension 4. On définit un nouveau type, pour les vecteurs réels de dimension 4, par

```
typedef float vec[4];
```

et on peut écrire ensuite

```
vec a;
```

```
#include <stdio.h>

struct gauss {
  int x;
  int y;
};

typedef struct gauss Gauss;

void imprimer(Gauss z) {
  if (z.y > 0) printf("%3d + %di\n", z.x, z.y);
  else if (z.y < 0) printf("%3d - %di\n", z.x, -z.y);
  else printf("%3d\n", z.x);
}

Gauss plus (Gauss u, Gauss v) {
  Gauss z;
  z.x=u.x+v.x;
  z.y= u.y+v.y;
  return z;
}

void pplus (Gauss u, Gauss v, Gauss *z) {
  z->x = u.x+v.x;
  z->y = u.y+v.y;
}

main() {
  Gauss a={1,0}, b={0,-123}, t, w;
  imprimer(a); imprimer(b);
  imprimer(plus(a,b));
  pplus(a,b,&t); imprimer(t);
  w = t; imprimer(w);
}
```