INF 111, L1, 2004/2005

Septembre 2004

Quelques rappels pour être plus efficace en TP

Laure Gonnord
`http://www-verimag.imag.fr/~gonnord/`

1 La syntaxe

1.1 Un programme

Syntaxe Un programme a cette allure :

```
#include <stdio.h>
#define PI 3.14

/*ici définition de fonctions */

int main (void)
{
  instruction1;
  instruction2;
  ...

  return 0;
}
```

Remarques

- Les accolades `{` et `}` définissent le *corps* (l'intérieur) du programme. Donc il n'y a RIEN entre l'en-tête `int main(void)` et l'accolade ouvrante.
- Les `#include` et les `#define` sont au tout début, et il n'y a PAS de point-virgule.
- Les fonctions utilisées dans le programme principal (c'est pour ça qu'il s'appelle `main`!) sont écrites dans la partie du fichier située avant la définition du programme principal (le `int main (void)` (en tout cas, pour vous!).
- La fonction principale retourne TOUJOURS `0` (zéro).
- La fonction principale n'aura jamais de paramètres (d'où le `void`), c'est possible mais nous n'en mettrons JAMAIS.
- Un fichier `.c` contient toujours une fonction principale OBLIGATOIREMENT nommée `main` et située à la toute fin du fichier. Le crochet fermant de la fonction `main` (qui dit "ceci est la fin de la fonction main") signifie donc "c'est la fin du programme", et donc est le dernier caractère du programme
- Seule la fonction `main` est exécutée, le reste du programme n'est exécuté que si on y fait appel (en particulier, les fonctions) à l'INTERIEUR de la fonction `main`.

1.2 Une fonction

Syntaxe Une fonction a cette allure :

```
double toto(char c,int b)
{
  double res,titi;

/*calcul du résultat de la fonction, stockage dans res*/

  return (res);
}
```

Ici :

- `toto` est le nom de la fonction. C'est celui qui servira à l'appeler ("appel de fonction").
- `double` est le *type de sortie* de la fonction. Cela veut dire que le résultat de la fonction `toto` est forcément un réel (remarquez qu'on le sait à l'avance).
- la fonction `toto` prend en *argument* un caractère `c` et un entier `b`. Ces arguments (ou paramètres) sont supposés CONNUS à l'intérieur de la fonction, on peut donc (et en général on doit) utiliser la valeur de ces paramètres pour calculer le résultat de la fonction. ATTENTION on ne doit pas redéclarer ces variables, elles existent.
- la variable `res` est une *variable auxiliaire*. C'est une variable *fraîche*, c'est-à-dire une nouvelle variable. Elle n'a donc pas de valeur, et il faudra bien l'initialiser (c'est-à-dire lui donner une valeur initiale) quelque part. Il en est de même pour la variable `titi`.

Remarques

- Une fonction ne retourne (ne renvoie) qu'une seule valeur (au plus, ce qu'on verra plus tard.). Il est impossible de renvoyer un couple d'entiers par exemple.
- Les variables auxiliaires n'existent ("ne vivent") qu'à l'intérieur de la fonction. En particulier, la valeur de la variable `titi` est perdue à jamais après la fin de la fonction `toto` Le nom lui-même (`titi`) est réutilisable, mais en aucun cas il ne s'agira de la même variable. Idem pour la variable res.
- les variables auxiliaires sont toujours déclarées au début de la fonction/du programme. On peut le faire ailleurs, mais ce n'est pas lisible.

Appels Comment se fait l'appel à la fonction ? (comment utiliser la fonction `toto`) :

- On appelle la fonction par exemple à l'intérieur de la fonction principale, de la façon suivante :
  `d = toto(cc,bb);`
  POURVU QUE :
  - `d` est déclarée et est un `double`.
  - `cc` est déclarée, est un caractère, et a une VALEUR. Cette valeur a par exemple été donnée par l'utilisateur à l'aide d'un `scanf`, ou alors elle a elle-même été calculée avant dans le programme.
  - `bb` est déclarée, est un entier, et a une VALEUR.
- On peut utiliser une fonction à l'intérieur d'une autre fonction en utilisant des appels comme décrit plus haut, pourvu que la fonction qu'on appelle existe, c'est-à-dire que son code est placé plus haut dans le fichier `.c`.

ATTENTION Bien distinguer la DÉCLARATION ("Soit une fonction dont le code est le suivant, son nom est etc") d'une fonction de son APPEL ("cette fonction existe, je vais l'utiliser"). En particulier, une fonction s'appelle avec `toto('a',5)` et PAS avec `toto(char 'a', int 5)`

1.3 Le if-then-else

Syntaxe

```
if (condition)
   {
      instruction1;
      instruction2;
      instruction3;
   }
else
   {
      instruction1;
      instruction2;
      instruction3;
      instruction4;
   };
```

Remarques

- Bien remarquer les PARENTHÈSES autour de la condition, et les ACCOLADES autour des actions (instructions).
- La condition est une condition de type booléen, c'est à dire que les conditions ont une valeur VRAI ou FAUX au moment où l'on exécute l'instruction `if`. La condition peut-être par exemple `x>=0`, `y==3`, `a=='c'` (si `a` est un caractère), ou bien une combinaison de conditions : `x>=0 || y==3` est une condition, qui est vraie dans le cas où l'une AU MOINS des deux conditions "x positif" et "y est égal à 3, est vraie.
- Il y a possibilité de s'affranchir des crochets lorsqu'il y a qu'une seule instruction dans le "then" ou dans le "else", mais je ne veux pas en entendre parler.
- Bien remarquer que le if-then-else est lui-même une instruction, d'où le point-virgule à la fin.

1.4 La boucle for

Syntaxe

```
for (i=depart;i<=arrivee;i=i+inc)
    {
      instruction1;
      instruction2;
      ...
    };
```

Avec `depart`, `arrivee`, `inc` des entiers fixés et `i` qui existe avant.

Ça fait quoi ? La variable `i` prend la valeur `depart` (un entier !), réalise la suite d'instructions à l'intérieur du corps de la boucle (dans l'ordre), puis ajoute `inc` à `i`. Si `i` satisfait la contrainte de boucle `i<=arrivee`, alors on continue, c'est-à-dire, on recommence la suite d'instructions du corps de la boucle ; sinon, on a fini l'instruction `for`, et on fait la suite. . . .

Remarques

- Comme pour le `if`, les PARENTHÈSES et les ACCOLADES, ce n'est pas pareil.
- La boucle `for` est elle même une instruction, d'où le point-virgule à la fin (pour vous !)
- La variable `i` dite "variable de boucle" doit être déclarée avant la boucle.

- La variable de boucle peut être utilisée à l'intérieur de la boucle (ou pas), mais attention on ne DOIT ABSOLUMENT PAS modifier la valeur de `i` à l'intérieur de ladite boucle.
- En général, `inc` vaut 1 ou -1. Pour des raisons de simplicité, on s'INTERDIT toute autre valeur de `inc`.
- Attention à la cohérence de la boucle. Si on fait une boucle croissante (`inc` vaut 1) alors on fera attention à avoir `depart` $\leq$ `arrivee`, si on fait une boucle décroissante, c'est le contraire.
- La boucle `for` est une instruction comme une autre, en particulier on peut mettre des boucles `for` à l'intérieur de boucles `for`, etc.
- On peut s'affranchir du point-virgule à la fin, mais si vous oubliez ensuite des points-virgules ailleurs, ne n'est pas mon problème.

1.5 La boucle while

Syntaxe

```
while (condition)
    {
      instruction1;
      instruction2;
      ...
    };
```

Avec `condition` une condition de type booléen (voir le paragraphe sur le `if`).

Ça fait quoi ? A l'entrée de la boucle, on regarde si la condition est vraie, si oui on va exécuter une fois le corps de la boucle (c'est-à-dire la suite d'instructions de la boucle) ; sinon on a fini. A la fin de la liste d'instructions on regarde si la condition est toujours vraie, si oui on refait la liste d'instructions, sinon on a fini. Bien remarquer que "on a fini" signifie que l'on passe à l'instruction suivante (celle qui est située juste après le crochet fermant).

Remarques
- Les mêmes que pour la boucle `for`.
- CONTRAIREMENT à la variable de boucle `for`, les variables qui interviennent dans la condition booléenne peuvent (et dans certains cas doivent) être modifiés à l'intérieur de la boucle. En particulier, si on veut faire une boucle descendante de 1 en 1 il ne faudra pas oublier `i=i-1` à chaque tour, si `i` apparaît dans la condition : `while(i>0){...}`

2 Création du programme, compilation

Comment écrire un programme C nommé `pair` ?

1. Créer un répertoire au nom du TP
2. Créer un fichier `pair.c`, par exemple en tapant `nedit pair.c`, puis en faisant `OK` à la demande de confirmation de création du fichier.
3. Taper le squelette du programme AVANT DE FAIRE l'EXO, vous devez savoir les yeux fermés écrire :

```
#include <stdio.h>

int main (void)
  {

     return 0;
  }
```

4. Compiler avec `gcc -Wall -o pair pair.c`

5. Corriger la PREMIÈRE erreur dans la liste
6. Compiler avec `gcc -Wall -o pair pair.c`
7. Corriger la première erreur
8. Compiler, corriger, ...jusqu'à ce que la compilation redonne la main "sans rien dire "
9. Exécuter le programme (taper `pair` dans le terminal), et tester plusieurs fois, avec des valeurs différentes des variables (choisies judicieusement)

N'oubliez pas le `#include<stdio.h>`, indentez-bien. Compilez avec `gcc -Wall -o toto toto.c`, et donnez des noms intelligents aux noms de fichiers [1] ainsi qu'aux noms de variables.

3 "Madame, ça marche pas !"

Il y a une différence entre `error` et `warning`. Si il y a UNE SEULE erreur, le programme exécutable n'est pas construit. Par contre, si il n'y a que des "warnings", le programme est construit mais son exécution n'est pas garantie. On s'attachera à enlever tous les warnings avant la première exécution du programme.

3.1 Trucs de base

1. As-tu bien appelé ton fichier `exo.c` (et pas `exo` tout court) ?
2. Tu compiles bien le bon fichier au bon endroit (dans ton HOME, de préférence) ?
3. C'est `gcc -o` et pas `gcc-o` tout collé, sinon ça répond "command not found".
4. Le truc que tu compiles, c'est bien la dernière version de ton exo (qui a bien été sauvegardée) ?

3.2 Les erreurs les plus fréquentes

1. `'#include' expects "FILENAME" or <FILENAME>`. Les inclusions de bibliothèques ne doivent pas avoir de point-virgule à la fin. Idem pour les `#define` (définition de constantes).
2. `Syntax error` : erreur de syntaxe, c'est-à-dire qu'il manque un point-virgule, un crochet, une parenthèse (peut-être sur la ligne précédente, voire plus haut !).
3. `implicit declaration of function '...'` : La fonction appelée n'existe pas. Elle n'a pas été écrite plus haut, que ce soit dans le fichier ou dans les `#include`.
4. `warning: too many arguments for format` : dans le `printf`, il doit y avoir autant de `%` que de variables. Par exemple `printf("toto")` est bon mais pas `printf("toto",x)`, la bonne version étant `printf("toto %d",x)`.
5. `warning: format argument is not a pointer (arg 2)` : il manque le `&` devant la variable du `scanf`.
6. `warning: passing arg 1 of 'printf' makes pointer from integer without a cast` : dans le `printf`, ne pas oublier la chaîne de mise en forme (la chaîne qui comprend les `%d`, `%f`, ...)
7. `parse error before ...` : erreur de "parsing", c'est plus grave qu'une erreur de syntaxe, cela signifie qu'il y a une erreur dans la mise en page des blocs, des fonctions, du programme principale. On a une telle erreur par exemple lorsqu'on appelle une fonction de la mauvaise manière (voir la section "fonctions").
8. `warning: unused variable 'i'` : la variable `i` a été déclarée, mais on ne s'en est pas servi.
9. `unterminated string or character constant` : est-ce que la chaîne du `printf` a bien été fermée ?
10. `warning: left-hand operand of comma expression has no effect` : un virgule au lieu d'un point-virgule ?
11. `warning: control reaches end of non-void function` : il y a un point virgule qui arrive un peu tôt.

[1]Faites ce que je dis, pas ce que je fais :-)

3.3 Ça compile, mais ça fait pas ce que l'on veut

1. Ça boucle : n'y-aurait-il pas une boucle `while` dont la condition est toujours vraie ?
2. Le résultat n'est pas celui escompté : il faut chercher dans le programme, éventuellement en imprimant des résultats intermédiaires supplémentaires. Cela s'appelle "débugger au printf".

4 Idées sur la conception d'un programme

- Essayer d'écrire un programme sans l'avoir conçu sur un papier, c'est perdu d'avance.
- On décrit d'abord sommairement (en français) sur son papier, l'algorithme que l'on va utiliser pour résoudre le problème, ensuite on réécrit le même en détaillant plus. Enfin on traduit en anglais/C.
- Rendre le source sans explication ce n'est pas faire l'exo.
- Écrire une fonction principale de 50 lignes, c'est mal.
- Si le programme n'est pas indenté de façon intelligente, il est illisible, donc non lu, donc ne vaut rien.
- On teste les fonctions/procédures auxiliaires sur des petits exemples, surtout si il y a beaucoup de code à écrire.
- **IMPORTANT** Un programme/ une fonction ne s'écrit pas de haut en bas en un seul trait, au contraire c'est un continuel va-et-vient entre les différentes lignes :
  - Déjà, on ferme l'accolade dès qu'on a mis l'accolade ouvrante (c'est aussi valable pour les parenthèses). On ouvre l'accolade, on appuie quelques fois sur ENTRÉE, puis on ferme l'accolade (au même niveau).
  - Lorsqu'on fait une boucle, on commence par écrire les lignes que l'on veut répéter, et ensuite on englobe le tout dans un for/un while.
  - On écrit d'abord le main avec les fonctions auxiliaires qui nous viennent à l'esprit, et après on écrit lesdites fonctions auxilaires [2].
  - Idem pour les variables, on écrit les tous les `i`, `j`, `tmp` qui nous chantent, et ensuite on les déclare.

[2]Mais "avant" dans le texte !