Cahier de TD n° 4 corrigé

THEME 1 : DEFINITION DE FONCTIONS, VRAI/FAUX 1

QUELQUES DEFINITIONS 1
VRAI/FAUX........ 1

THEME 2 : FONCTIONS SIMPLES, DEFINITIONS ET APPELS........ 2

EXEMPLES DE DEFINITION ET D'APPELS DE FONCTION : 2
ARGUMENTS / PARAMETRES / VARIABLES LOCALES 6
UN PROGRAMME MAL NOMME 8

THEME 3 : FONCTIONS ET TABLEAUX STATIQUES/DYNAMIQUES 10

FONCTIONS UTILITAIRES 10
RETOURNER UN TABLEAU ?........ 11
CAS D'UN TABLEAU STATIQUE........ 11
CAS D'UN TABLEAU DYNAMIQUE 12

Thème 1 : Définition de fonctions, vrai/faux

Quelques définitions

Parmi les entêtes qui suivent, lesquelles sont incorrectes et pourquoi ?

`function 3x(reel x, reel y, reel z)` **incorrect : on ne précise ni entrée, ni sortie.**

`fonction racine_car(entree : reel → sortie : reel)` **incorrect : l'entrée n'a pas de nom**

`fonction racine(entree : reel toto → sortie : reel)` **correct**

`fonction foo(entree : entier a,b,c → sortie : entier,reel)` **incorrect : il y a deux sorties, les types des entrées (paramètres) doivent être rappelés pour chaque entrée**

`fonction bar(→ sortie : caractere)` **: correct**

`fonction x3Y_fT2Aé(entree : entier Azrz4_)` **correct**

`fonction gauss(entree : reel x, entier sigma → sortie : reel)`**correct**

VRAI/FAUX

- Un argument est une entrée pour la définition de la fonction; **FAUX, lors de la définition, une entrée est une paramètre**
- Un argument est une valeur donnée par un programme lors de l'appel d'une fonction; **VRAI**
- Une fonction a au moins une sortie; **FAUX**
- Une fonction a au plus une sortie; **VRAI**
- Il faut écrire autant d'exemplaires d'une fonction que de nombres de fois où on veut l'utiliser; **FAUX : un seul exemplaire appelé plusieurs fois.**
- Une fonction rend un programme plus lisible; **VRAI**
- Un programme avec des fonctions est plus dur à maintenir qu'un programme sans fonctions (maintenir = faire la chasse aux bugs); **FAUX, c'est même exactement le contraire**
- On doit obligatoirement appeler une fonction que l'on définit; **FAUX, on peut ne pas l'utiliser.**
- On doit obligatoirement définir une fonction que l'on appelle; **VRAI**

- `afficher` est une fonction; **VRAI**
- `while` est une fonction. **FAUX, c'est un mot clef du langage C**

Thème 2 : fonctions simples, définitions et appels

exemples de définition et d'appels de fonction :

Ecrivez les programmes suivants avec des fonctions. Le programme principal devra appeler la ou les fonctions définies au moins 2 fois.

Pour la correction : je fournis juste les définitions de fonction/

- Programme faisant la somme de deux entiers;

```
fonction somme_ent(entree : entier a1, entier a2 → sortie :entier)
{
      retourner a1+a2;
}
```

- Programme calculant la moyenne de deux entiers;

```
fonction moy_ent(entree : entier a1, entier a2 → sortie :reel)
{
      reel moyenne;
      moyenne ← (a1+a2)/2.0;     // attention à éviter la division
                                 // entière
      retourner moyenne;
}
```

- Programme proposant un affichage propre : on fournit un nombre entier, par exemple qui indique le nombre N d'€ que vous avez gagné dans un jeu. Selon la valeur de N, qui peut être nulle, égale à 1 ou supérieure, le programme affichera un message tenant compte de cette valeur de N, notamment pour gérér le cas singulier/pluriel : on veut éviter d'afficher, par exemple, les messages :
  "Vous avez gagné 1 euro**s**" (car on doit mettre euro au singulier ici).

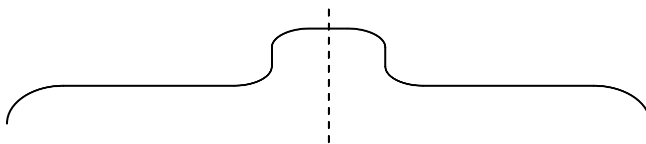
```
fonction aff_propre(entree : entier nb)
{
      si (nb=0) alors
      {
            afficher("vous n'avez rien gagne");
      }
      sinon si (nb=1) alors
      {
            afficher("vous avez gagne 1 euro");
      }
      sinon
      {
```

```
            afficher("vous avez gagne ",nb, " euros");
      }
}
```

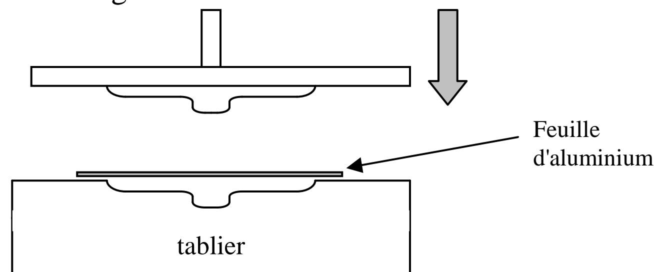
Plusieurs fonctions pour simplifier un calcul complexe

Le processus industriel d'emboutissage consiste à mettre en forme des pièces de forme sophistiquée à partir d'un matériau souple (par exemple l'aluminium en feuilles minces) à l'aide d'une presse. La feuille d'aluminium est présentée sur le tablier de la presse, et la partie mobile de la presse supporte un modèle de la forme à emboutir.

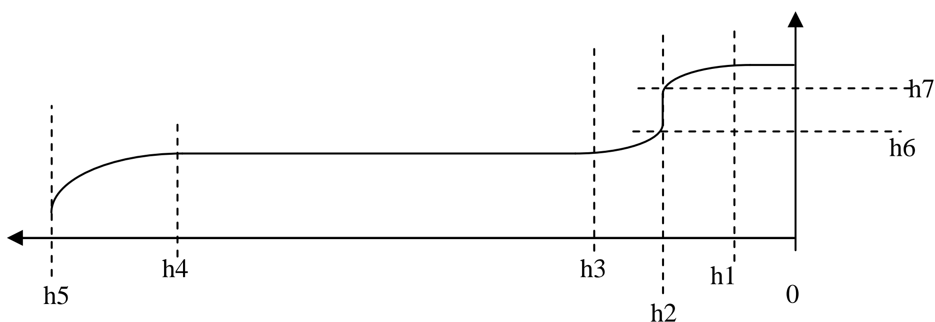
Exemple : pour emboutir un couvercle dont la forme est :



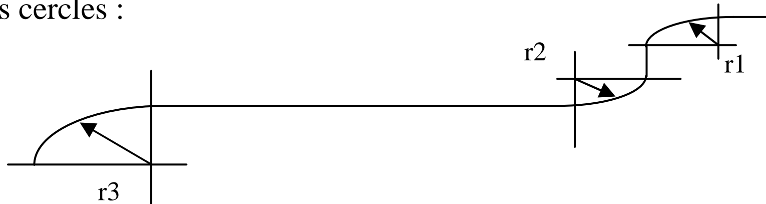
La presse aura cette configuration :



Cette forme est à symétrie axiale, et est composée de 6 parties, dont on vous donne les dimensions en mm.



En ce qui concerne les parties en arc de cercle (ce sont effectivement des morceaux de cercle, même si elles apparaissent sur le schéma comme des arc d'ellipse), on indique les rayons des cercles :



Ecrire un programme qui calcule la surface de chacune des 6 parties de la forme et qui indique la surface totale du couvercle.

Principe du calcul : l'aire A du couvercle est égal à $\pi.r_f^{\,2}$

Où r_f est le rayon, que l'on calcule à partir du profil de la forme emboutie. Ce rayon r_f est calculé en 6 parties :

$$rf = h1 + \frac{\pi}{2} r1 + (h7 - h6) + \frac{\pi}{2} r2 + (h4 - h3) + \frac{\pi}{2} r3$$

(les arcs de cercle sont forcément des quarts de cercle, car il faut assurer la tangence entre les parties linéaires horizontales et verticales).

On cherche maintenant à calculer les surfaces individuelles des parties : chaque partie est en fait, un anneau concentrique. Il faut donc écrire une fonction calculant l'aire d'un anneau concentrique : pour un anneau concentrique de rayon interne ri et de rayon externe re, la surface à calculer est :

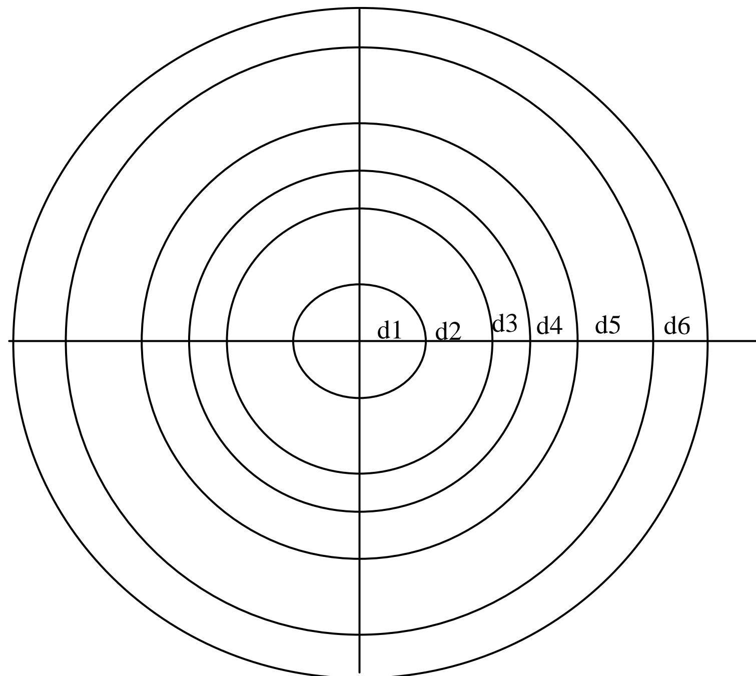
Chaque fonction calculera donc : $\pi.r_e^{\,2} - \pi.r_i^{\,2}$

```
fonction aire(entree : reel r_e, reel r_i → sortie : reel)
{
      reel resultat;

      resultat = M_PI*(r_e*r_e - r_i * r_i);

      retourner resultat;
}
```

sous forme dépliée (avant emboutissage), le couvercle se présente ainsi (vue de dessus)



Où les dimensions correspondant à d1,d2,…,d6 valent respectivement :

h1, $\frac{\pi}{2}r1$, h7-h6, $\frac{\pi}{2}r2$, h4-h3, $\frac{\pi}{2}r3$

on calcule donc S, la surface totale, de la manière suivante :

```
programme surface_couvercle
reel surf_tot;
reel dim;
// h1 à H7 et r1 à r3 connus et définis
surf_tot ← aire(0,h1);
dim ← h1;
surf_tot ← surf_tot+aire(dim, dim+(M_PI*r1/2.0));
dim ← dim++(M_PI*r1/2.0);
surf_tot ← surf_tot+aire(dim, dim+(h4-h3));
etc…
```

Sachant que la feuille d'aluminium est rectangulaire, et qu'on garde une marge de sécurité de *s* mm au bord de la feuille, quelle sera la surface de la chute ?

La feuille est carré, son côté doit correspondre au diamètre du couvercle + 2 fois la marge de sécurité : il faut donc écrire une fonction calculant rf.

La surface de la feuille est donc A_f **:** $(2.(r_f+s))^2$

La surface du couvercle est de Ac: $\pi.r_f^{\ 2}$

La surface de chutes est égale à la différence entre les deux

Ecrire une fonction qui calcule la surface totale des chutes pour l'emboutissage d'un nombre *N* de couvercles.

Ecrire une fonction calculant le pourcentage de chute pour un nombre *N* de couvercles.

Pourcentage : $N.(A_f\text{-}A_s) / N.A_f = (A_f - A_s) / A_f$

arguments / paramètres / variables locales

Dans le programme suivant, indiquez où sont les arguments, les paramètres, les appels et les définitions des fonctions :

```
fonction gilb_1(entree : entier g1, entier g2 → sortie : entier)
{
      entier g_temp;

      g_temp ← g1+g2;
      g2 ← g_temp - g2;
      g1 ← g1 - g_temp + g1;
      g2 ← g1 - g2;

      retourner(g_temp);
}

fonction gilb_2(entree : reel g_1, caractere g_3 → sortie : reel)
{
      entier g_temp;

      g_temp ← g_1;

      retourner(g_1/g_temp);
}

programme gilb_fonc
{
      entier g_1, g_2, g_cpt;
      reel g_x;

      afficher("g_entrez deux g_entiers :");
      saisir(g_2);
      saisir(g_1);

      gilb_2(1.34543,'X');

      afficher("g_resultat :",gilb_1(g_2-g_1,g_1-g_2));

      g_1 ← gilb_1(g_1,g_2);
      afficher("g_resultat 2 : ",gilb_1(gilb_1(g_1,g_2),g_2));

      afficher("entrez le g_nombre de g_points :");
      saisir(g_1);
```

```
        g_x ← 0.0:

        pour g_cpt de 0 a g_1
        {
                g_x ← 1.0 + g_cpt * (5.0/g_1);
                afficher(g_x,"  ", gilb_2(g_x,'#'),"\n");
        }
}
```

rappel : les paramètres sont les entrées lors des définitions de fonctions.

Les arguments sont les valeurs fournies lors de l'appel de la fonction.

La définition d'une fonction est son écriture.

L'appel de fonction est son utilisation : attention, il y a comme appel de fonction les appels à gilb_1 et gilb_2, bien entendu, mais aussi les appels à : afficher() et saisir().

Gilbert vous dit que toutes les variables sont locales. Etes-vous d'accord avec lui ?

Oui, toutes les variables sont locales.

Que fait ce programme ?

La fonction gilb_1, malgré son apparence compliquée, ne retourne que la somme de ses deux arguments.

La fonction gilb_2 retourne l'argument réel divisé par sa partie entière, l'argument caractère ne joue aucun rôle.

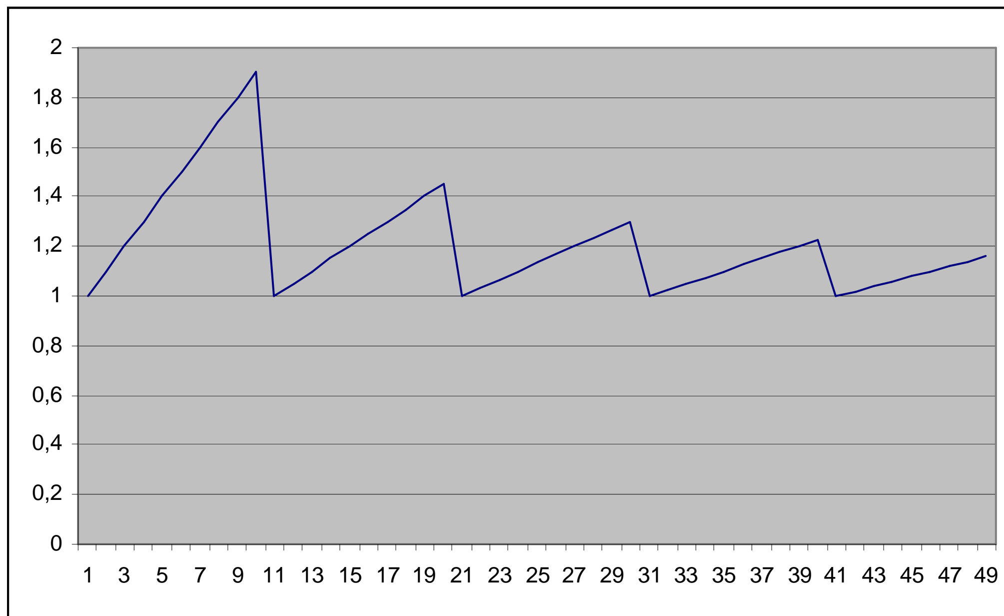
Premier appel à gilb_1 pour l'affichage : g_temp vaut 0, donc affiche 0

Deuxième appel à gilb_1 : g_temp contient la somme des deux arguments : donc g_1 reçoit g_1+g_2.

Troisième appel à gilb_1 : affiche la somme : g_1+g_2+g_2 (attention g_1 a été modifié par le deuxième appel à gilb_1).

Quel serait l'allure de la courbe dessinée à partir des points calculés à la fin du programme ?

C'est la fonction x/E[x], qui a l'allure suivante :



Un programme mal nommé

Donnez les valeurs des variables au fur et à mesure du déroulement du programme suivant : que constatez-vous ?

```
fonction echange(entree : entier a, entier b)
{
        entier excg;
        excg ← a;
        a ← b;
        b ← exg;

        afficher("a = ",a," et b = ",b,"\n");
}

programme swap

entier a,b;

a ← 5;
b ← 2;
afficher("a = ",a," et b = ",b,"\n");
echange(b,a);
afficher("a = ",a," et b = ",b,"\n");
echange(a,b);
afficher("a = ",a," et b = ",b,"\n");
```

C'est l'exemple classique d'une fonction échange pour lesquels les paramètres sont passés par valeur (il n'y a pas de pointeurs). Donc cette fonction ne change absolument pas les valeurs des variables a et b du programme principal !

il affiche donc :

a=5, b=2 (prog)

a=5, b=2 (fonction)

a=5, b=2 (prog)

a=2, b=5 (fonction)

a=5, b=2 (prog)

Thème 3 : fonctions et tableaux statiques/dynamiques

Fonctions utilitaires

Note :vous pourrez réutiliser ces fonctions standard à de nombreuses occasions pour vos projets futurs !

Ecrire une fonction qui affiche tous les éléments d'un tableau dont le type est connu (à vous de choisir le type). Précisez quelles sont les entrées et la sortie de cette fonction

```
fonction aff_tab(entree : reel tab[], entier util) (pas de sortie)
{
      entier cpt;

      pour cpt de 0 à util faire
      {
            afficher(tab[cpt]," ");
      }
      afficher("\n");
}
```

Ecrire une fonction qui effectue la saisie d'un certain nombre d'éléments à ranger dans un tableau dont le type est connu. Précisez quelles sont les entrées et la sortie de cette fonction.

On fournit : le tableau, la taille maximum, et on retourne la taille utile.

```
fonction saisi_tab(entree : reel tab[], entier max → sortie :
entier)
{
      entier cpt;
      caractere rep;
      cpt ← 0;

      faire
      {
            afficher("entrez une valeur");
            saisir(tab[cpt]);
            cpt ← cpt+1;
            afficher("autre saisie ? (o/n) :");
            saisir(rep);
      }
      tant que ((cpt < max) et (rep='o'));

      retourner(cpt);

}
```

Rappelez pourquoi il n'est pas nécessaire de retourner un tableau dans une fonction qui modifie les éléments stockés dans le tableau.

Car le tableau est une adresse.

Ecrire une fonction qui effectue le tri d'un tableau par ordre croissant ou décroissant, le choix de l'ordre de tri se fait par l'intermédiaire d'un paramètre `ordre_tri`. Précisez quelles sont les entrées et la sortie de cette fonction.

```
fonction tri(entree : entier tab[], entier util, entier ordre)
{
	entier temp, cpt1, cpt2;
	// ordre = 0 : tri croissant, ordre = 1 : tri decroissant

	pour cpt1 de 0 à util-1
	{
		pour cpt2 de 0 à util-2
		{
			si (ordre=0 et tab[cpt2]>tab[cpt2+1]) ou (ordre=1 et
tab[cpt2]<tab[cpt2+1]) alors
			{
				temp ← tab[cpt2];
				tab[cpt2]←tab[cpt2+1];
				tab[cpt2+1] ← temp;
			}
		}
	}
}
```

Retourner un tableau ?

Cas d'un tableau statique

Une fonction peut parfois avoir besoin de retourner un tableau, par exemple une fonction qui effectue une copie d'un tableau d'entiers, ou de réels. Que se passe-t-il exactement dans ce cas ? Nous allons tenter de le déterminer en regardant le programme suivant.

```
fonction copie_tab(entree : entier tab[], entier util → sortie :
entier [])
{
	entier tab_res[100];
	entier cpt;

	pour cpt de 0 à util-1
	{
		tab_res[cpt] ← tab[cpt];
	}

	retourner tab_res;
}
```

```
fonction affich_tab(entree : entier tab[], entier t_ut)
{
      entier cpt;

      pour cpt de 0 à t_ut-1
      {
            afficher(tab[cpt]," ");
      }
      afficher("\n");
}

fonction principale()
{
      entier tablo[100] ← {1,2,3,4,5,6,7,8};
      entier util;
      entier *resultat;

      util ← 8;

      resultat ← copie_tab(tablo, util);

      affich_tab(resultat, util);
}
```

question 1) Quel est le type de la sortie de la fonction `copie_tab` ?

question 2) Pourquoi doit-on utiliser un tableau dynamique pour la variable `resultat` du programme principal et non une définition statique telle que : `entier resultat[100]` ?

question 3) Que vaut, dans la fonction `copie_tab`, la variable `tab_res` ? (vous pouvez faire une hypothèse sur la valeur numérique de cette variable).

question 4) Que vaut, dans le programme principal, la variable `tab_res` ?

question 5) Que vaut la variable `resultat` du programme principal après l'appel à la fonction `copie_tab` ?

question 6) Que trouve-t-on en mémoire à l'adresse stockée dans la variable `resultat` du programme principal ?

question 7) Le programme affichera-t-il les valeurs du tableau ou affichera-t-il un message d'erreur ?

cas d'un tableau dynamique

voici une nouvelle version de la fonction copie_tab, fonctionnant avec un tableau dynamique. Répondez de nouveau aux questions 4 à 7 de l'exercice précédent, et indiquez les instructions à ajouter éventuellement dans et/ou à la fin du programme principal pour adapter le programme à la nouvelle version de cette fonction.

```
fonction copie_tab(entree : entier tab[], entier util → sortie :
entier *)
{
      entier *tab_res;
      entier cpt;

      tab_res ← reservation(util entier);

      si (tab_res ≠ NULL) alors
      {
            pour cpt de 0 à util-1
            {
                  tab_res[cpt] ← tab[cpt];
            }
      }

      retourner tab_res;
}
```