LIFAP1 – TD 4 : Passage de paramètres

Objectifs : Comprendre la différence entre les modes de passage des paramètres : donnée ou donnée / résultat.
Comprendre la différence entre paramètres formels et paramètres effectifs

Recommandations :
Pour chacun des algorithmes que vous écrirez vous préciserez le mode de passage des paramètres (**donnée** ou **donnée / résultat**) et vous écrirez le programme principal appelant les sous-programmes que vous aurez écrits.

Soit le programme suivant :

```
#include <iostream>
using namespace std ;

void mystere (int a, int b, int &c, int d)
{c=a+b;
 d=a*b;
}

int main (void)
{int e,f,g,h;
 cout<<"donnez une valeur";
 cin>>e;
 cout<<"donnez une valeur";
 cin>>f;
 mystere(e,f,g,h);
 cout<<" valeur "<<g<<" valeur :"<<h<<endl;
 return 0;
}
```

1. Identifiez et notez :
   a. le(s) paramètre(s) formel(s) / le(s) paramètre(s) effectif(s)
   b. le(s) paramètre(s) en donnée / le(s) paramètre(s) en donnée / résultat
   c. Qu'est censé faire ce programme ?
   d. Quelle(s) modification(s) faudrait-il apporter pour obtenir un résultat plus logique ?

Profitez de ce premier exercice pour faire quelques rappels de cours en donnant les définitions…

Rappels de cours (définition) :

Paramètre formel : variable utilisée dans le corps du sous-programme qui reçoit une valeur de l'extérieur (ils font partie de la description de la fonction)
Paramètre effectif : il s'agit de la variable (ou valeur) fournie lors de l'appel du sous-programme (valeurs fournies pour utiliser la fonction et valeurs renvoyées)
Copie de la valeur du paramètre effectif vers le paramètre formel correspondant lors de l'appel.
Paramètres formel et effectif ont des noms différents

Données (passage par valeur) : le sous-programme dispose d'une copie de la valeur.
Il peut la modifier, mais l'information initiale dans le code appelant n'est pas affectée par ces modifications.
Syntaxe en C/C++ : type nom ;

Résultats ou données / résultats (passage par adresse) : le sous-programme dispose d'une information lui permettant d'accéder en mémoire à la valeur que le code appelant cherche à lui transmettre. Il peut alors modifier cette valeur, le code appelant aura accès aux modifications faites sur la valeur.
Syntaxe en C/C++ : type & nom ;

Paramètres formels : a, b, c et d
Paramètres effectifs : e, f, g, h
Paramètres en donnée : a, b, d
Paramètres en donnée / résultat : c

Le programme est censé calculer et retourner la somme et le produit de deux variables a et b. La somme est stockée dans la variable c et le produit dans la variable d. Pour obtenir le résultat attendu, il faut passer le paramètre formel d en donnée / résultat sinon la valeur calculée dans la procédure est perdue définitivement.

2. Écrivez l'algorithme d'une procédure effectuant la permutation circulaire de trois variables : a=5 b=8 et c=2 donne après exécution : a=2 b=5 et c=8

```
Procédure permutation_circulaire (a : entier, b : entier, c : entier)
Précondition : aucune
Données / Résultats : a, b et c
Description : effectue la permutation circulaire des 3 variables a, b et c
Variable locale : tampon : entier
Début
  tampon ← c
  c← b
  b← a
  a← tampon
Fin permutation_circulaire

Appel :
Début
  Variables locales : v1, v2, v3 : entier
  Afficher ('première valeur')
  Saisir (v1)
  Afficher ('deuxième valeur')
  Saisir (v2)
  Afficher ('troisième valeur')
  Saisir (v3)
  permutation_circulaire (v1,v2,v3)
  Afficher ('nouvelles valeurs : ', v1, ' ', v2, ' ', v3)
Fin
```

3. Écrivez l'algorithme d'une procédure permettant d'effectuer la division euclidienne de deux entiers a et b. On appellera q le quotient et r le reste de cette division. On rappelle la formule de la division : a = b*q + r, avec r < b.

```
Procédure division_euclidienne (a : entier, b : entier, q : entier, r : entier)
Précondition : aucune
Données : a et b
Données / Résultats : q et r
Description : effectue la division euclidienne de a par b
Variable locale : aucune
Début
  q←0
  r ← a
```

```
  Tant que  (r >= b) faire
    q ←q+1
    r ←r-b
  Fin Tant que
Fin division_euclidienne

Appel :
Début
  Variables locales : v1, v2, quotient, reste : entiers
  Afficher (‘première valeur :’)
  Saisir (v1)
  Afficher (‘deuxième valeur :’)
  Saisir (v2)
  Division_euclidienne (v1,v2, quotient, reste)
  Afficher (v1, ‘ / ’, v2, ‘donne ’, quotient, ‘et reste ‘,reste)
Fin
```

Traduction en C :

```
void divisionEuclidienne(int a, int b, int &q, int &r)
{
  q=0;
  r=a;
  while(r>=b)
  {
    q=q+1;
    r=r-b;
  }
}
int main()
{
  int a, b, q, r;
  a=30;
  b=4;
  q=0;
  r=0;
  divisionEuclidienne(a, b, q, r);
  cout << "Quotient : " << q << " et reste: " << r <<endl;
  return 0;
}
```

4. Écrivez l’algorithme d’une fonction perimetre_cercle permettant de retourner le périmètre d’un cercle en fonction de son rayon (passé en paramètre). Écrivez ensuite une fonction aire_cercle qui retourne l’aire d’un cercle. On souhaite maintenant écrire un sous-programme (qui utilise les deux fonctions précédentes) permettant à partir du rayon d’un cercle de calculer son périmètre et sa surface. Écrivez l’entête de ce sous programme de deux manières différentes.

```
Fonction perimetre_cercle (r : entier) : entier
Précondition : r > 0
Données : r rayon du cercle
Résultats : perimetre du cercle
Variable locale : aucune
Début
  Retourner (2*3,14159 *r)
Fin perimetre_cercle

Appel :
Variables locales : rayon : entier
Afficher (‘donnez le rayon’)
```

```
Saisir (rayon)
Afficher(perimetre_cercle(rayon))

Fonction aire_cercle (r : entier)
Précondition : r > 0
Données : r rayon du cercle
Résultats : aire du cercle
Variable locale : aucune
Début
  Retourner (3,14159 * r * r)
Fin aire_cercle

Appel :
Variables locales : rayon : entier
Afficher ('donnez le rayon')
Saisir (rayon)
Afficher(aire_cercle(rayon))
```

Première version :

On fait une **procédure** et on intègre les deux résultats aux paramètres. Ils seront donc tous les deux passés en donnée / résultat puisque modifiés à l'intérieur du programme.

```
procedure perim_aire (r : entier, p : réel , a : réel)
Précondition : r > 0
Données : r rayon du cercle
Données / Résultats : p et a respectivement périmètre et aire du cercle de rayon r
Variable locale : aucune
Début
  p ← perimetre_cercle(r)
  a ← aire_cercle (r)
Fin perim_aire

Appel :
Variables locales : rayon : entier, peri, surf : réels
Afficher ('donnez le rayon')
Saisir (rayon)
perim_aire(rayon,peri,surf)
Afficher(peri, surf)
```

Deuxième version :

On fait une **fonction** qui retourne l'une ou l'autre des deux valeurs (périmètre ou aire) et on intègre l'autre résultat aux paramètres.

```
fonction perim_air2 (r : entier, p : réel) : reel
Précondition : r > 0
Données : r rayon du cercle
Données / Résultats : p périmètre du cercle de rayon r
Résultat : aire du cercle
Variable locale : aucune
Début
  p ← perimetre_cercle(r)
  retourner ( aire_cercle (r))
Fin perim_aire2

Appel :
Variables locales : rayon : entier, peri, surf : réels
Afficher ('donnez le rayon')
Saisir (rayon)
peri= perim_aire2(rayon, surf)
Afficher(peri, surf)
```

5. Écrivez l'algorithme d'une fonction qui à partir de deux entiers n et p calcule le nombre de combinaisons de p éléments pour un ensemble de n éléments. Rappel : $C_n^p = n ! / (p ! * (n-p) !)$.
   Transformez cette fonction en procédure puis traduisez en langage C.

   Pour cet exercice, on réutilisera la fonction factorielle du cours.

   Version fonction:

```
Fonction combinaison (n : entier, p : entier) : entier
Précondition : n>0 et n>p
Données : n et p
Données / Résultats : aucun
Résultat : combinaison
Variable locale : aucune
Début
  Retourner ((factorielle(n)/(factorielle(p)*factorielle(n-p)))
Fin combinaison

Appel :
Variables locales : n, p, comb : entiers
Afficher ('donnez les coefficients n et p :')
Saisir (n)
Saisir (p)
comb← combinaison(n, p)
Afficher(comb)
```

   Ici les paramètres formels et effectifs portent le même nom …. Juste pour montrer qu'on peu le faire quand même mais qu'il ne s'agit pas en mémoire de la même variable !!

   Version procédure :

```
Procédure combinaison (n : entier, p : entier, combin : entier)
Précondition : n>0 et n>p
Données : n et p
Données / Résultats : combin
Résultat : calcule le Cnp
Variable locale : aucune
Début
  combin ← (factorielle(n)/(factorielle(p)*factorielle(n-p))
Fin combinaison

Appel :
Variables locales : n,p, comb : entiers
Afficher ('donnez les coefficients n et p :')
Saisir (n)
Saisir (p)
combinaison(n, p,comb)
Afficher(comb)
```

   Traduction en C/C++ :

```
void combi(int n, int p, int &c)
{  c = ((factorielle(n)/(factorielle(p)*factorielle(n-p)) ; }
```

6. Un nombre entier est dit "doublon" si le produit de ses diviseurs est multiple de la somme de ses diviseurs.

Exemple : n = 6. Les diviseurs de n sont : 1, 2, 3, 6. La somme des diviseurs est 12 et le produit des diviseurs est 36 (= 3 * 12). Le produit des diviseurs de `n` est donc un multiple de la somme des diviseurs de `n` donc `n` est un nombre doublon.

a. Ecrire l'algorithme d'un sous-programme `somme_produit` permettant de calculer et "renvoyer" au programme principal la **somme des diviseurs et le produit des diviseurs** d'un nombre `n` passé en paramètres.

```
Procédure somme_produit (n : entier , s : entier, p : entier)
Préconditions : n>0
Données : n
Données / résultats : s,p
Description : calcule et "retourne" la somme et le produit des diviseurs de n
Variables locales : i : entier
Début
  s← 0
  p←1
  Pour i allant de 1 à n par pas de 1 faire
    Si (n modulo i) = 0 alors
                s←s+i
                p←p*i
    Fin si
  Fin pour
Fin
```

b. Ecrire l'algorithme d'une **fonction booléenne** `verifie_doublon` qui retourne vrai si un entier passé en paramètres est un doublon, faux sinon. On utilisera pour cela le sous-programme écrit dans la question précédente.

```
Fonction verifie_doublon (n : entier) : booléen
Préconditions : n>0
Données : n
Données / résultats : aucune
Résultat : booléen
Description : retourne vrai si n est un nombre doublon, faux sinon
Variables locales : som, prod : entier
Début
  somme_produit(n,som,prod)
  Retourner (prod modulo som)=0
Fin
```

Pour s'entraîner

1. Ecrire l'algorithme d'une procédure permettant à partir des trois coefficients a, b et c d'un polynôme du second degré, de calculer et retourner (si elles existent) les racines.