Chapitre 5

Fonctions II

5.1 Valeur de retour

Une fonction peut être appelée pour calculer une valeur. Par exemple, la fonction prédéfinie `abs()` calcule la valeur absolue d'un nombre. Pour transmettre la valeur calculée au programme appelant (flèche "sorties" dans la figure 4.1), on utilise dans le corps de la fonction l'instruction `return` suivie de la valeur à retourner (on dit aussi renvoyer). De plus, le type de la variable retournée doit figurer au début de l'en-tête ; c'est le **type de retour** de la fonction. Lorsqu'on ne renvoie pas de valeur, le type de retour est `void`.

Remarque : On ne peut renvoyer plus d'une valeur en C++. Nous verrons plus tard comment transmettre d'autres valeurs de sortie au programme appelant.

Voici un exemple de fonction retournant le périmètre d'un rectangle, de type `double` :

```
double perimetre_rectangle(double longueur, double largeur)
{
  double perimetre = 2 * (longueur + largeur);
  return perimetre;
}

int main()
{
  affichage("Le périmètre du rectangle est : ", perimetreRectangle(5.2,3.4), '\n');
  return 0;
}
```

Attention : Lorsque l'instruction `return` est exécutée, la fonction se termine : aucune instruction de la fonction n'est exécutée ensuite. L'instruction `return` peut être utilisée même sans renvoyer de valeur.

▶ Sur ce thème : **Exercices 1 et 2 à 4 du TD 5**

5.2 Fonction principale

Le programme `main` est lui-même une fonction, automatiquement appelée par le système d'exploitation à l'exécution du programme. La valeur de retour, de type `int`, est un code d'état d'exécution du programme qui peut être récupéré par le système. La valeur est 0 si l'exécution est correcte ; une valeur non nulle correspond à un code d'erreur.

Les autres fonctions doivent être définies **avant** la fonction `main`. Cela fait, on les appelle à partir du `main`. Plus généralement, une fonction peut appeler une autre fonction. Elle joue alors le rôle de programme appelant.

5.3 Définir une fonction

Lorsqu'on définit une fonction, il faut déterminer les paramètres, leur type, leur mode de passage et la valeur de retour avant de commencer à écrire le corps de la fonction. La création d'une fonction se fait donc en plusieurs étapes expliquées ici.

5.3.1 Étapes

Il est très important de déterminer les arguments nécessaires à la fonction et la valeur de retour. (Attention à ne pas confondre paramètres et variables locales.) Il faut ensuite choisir le mode de passage des paramètres, par copie ou par référence. *Seuls les paramètres correspondant à une sortie doivent être passés par référence* (on verra ce mode de passage ultérieurement). Avant de penser à écrire le corps de la fonction, il faut impérativement réfléchir aux informations nécessaires pour effectuer le traitement. La définition d'une fonction doit se faire en plusieurs étapes :

1. Que doit réellement faire cette fonction ?
2. Écrire l'en-tête. Comment nommer la fonction ? Quels sont ses paramètres ? Lesquels sont passés par copie, par référence ? Quels sont leurs types ? La valeur et le type de retour ?
3. Écrire un exemple d'appel.
4. Écrire le corps de la fonction.

Il est inutile de commencer à écrire le corps de la fonction si les autres étapes n'ont pas été franchies.

Remarque : Il est important de commenter une fonction lors de sa définition pour expliquer son utilisation (Que fait la fonction ? Quels sont ses paramètres ? quelle valeur retourne-t-elle ?). Ceci permet à d'autres programmeurs de pouvoir utiliser cette fonction sans devoir lire son code.

5.3.2 Exemple

On illustre la création de fonctions à l'aide d'un exemple. Une des interactions les plus courantes avec l'utilisateur consiste à lui demander confirmation d'une proposition (répondre par oui ou par non) et à contrôler que sa réponse est bien une des réponses attendues. C'est si fréquent que cela vaut la peine d'en faire une fonction.

Étape 1 La fonction doit permettre à l'utilisateur de confirmer ou d'infirmer une proposition. La fonction va donc devoir afficher un message puis lire la réponse saisie par l'utilisateur. Bien que cette fonction soit extrêmement simple, elle soulève plusieurs questions :

— Quelles sont les réponses possibles ? Comme on suppose que le programme est écrit en français, on choisit alors d'admettre comme réponses possibles : "oui", "non", "o", "n". De plus, les réponses peuvent être écrites avec des lettres minuscules et/ou majuscules.
— Que se passe-t-il si l'utilisateur ne saisit pas une des réponses permises ? On décide alors que l'on répète la saisie jusqu'à obtenir une réponse admissible.

Étape 2 Il faut alors répondre aux questions :

— *Comment nommer la fonction ?* On peut nommer cette fonction `confirmation`.
— *Quels sont ses paramètres ? Lesquels sont passés par copie, par référence ? Quels sont leurs types ? La valeur et le type de retour ?* Comme la saisie est faite à l'intérieur de la fonction, il n'y a pas besoin d'entrée. Il faut par contre que la fonction transmette la réponse de l'utilisateur, à savoir s'il confirme ou non. On peut alors décider qu'il n'y a aucun paramètre et une valeur retournée. Comme il n'y a que deux réponses effectives (l'utilisateur confirme ou infirme), on peut renvoyer un booléen.

— *On écrit l'en-tête et le commentaire :*

```
/*
 * Fonction demandant à l'utilisateur de répondre par oui ou non.
 * La saisie est répétée jusqu'à ce que l'utilisateur saisisse une des réponses permises :
 * "oui", "non", "o", "n", (majuscules et/ou minuscules).
 * La fonction renvoie true si l'utilisateur a répondu "oui" ou "o", et false sinon.
 */
bool confirmation ()
{
  //Corps de la fonction
}
```

Étape 3 Voici un exemple d'appel de la fonction `confirmation` :

```
int main()
{
  affichage("La terre est-elle ronde ?\n");
  if (confirmation())
    affichage("Bravo\n");
  else
    affichage("Il faut revoir les bases de l'astronomie...\n");
  return 0;
}
```

Étape 4 Corps de la fonction :

```
bool confirmation()
{
  string rep = "";
  while (rep!= "oui" && rep != "o" && rep != "non" && rep != "n")
  {
    affichage("Répondez par oui ou par non (o/n) : ");
    saisie(rep);
    rep = lower(rep); //On met la réponse en minuscules
  }

  if (rep=="oui" || rep=="o")
    return true;
  return false;
}
```

5.3.3 Tests de fonctions

Il est très important de tester les fonctions que l'on a écrites pour vérifier qu'elles ne comportent pas d'erreurs. Tester une fonction consiste à appeler cette fonction avec des valeurs de paramètres spécifiques, et de vérifier que le résultat de l'appel (par exemple la valeur retournée) est conforme à celui attendu.

Pour tester correctement une fonction, il faut effectuer plusieurs tests avec des valeurs de paramètres différentes, de manière à tester tous les comportements possibles de la fonction. Il faut aussi tester les cas limites (valeurs proches du changement de comportement de la fonction).

Par exemple, supposons que l'on veuille tester la fonction `int factorielle(int nb)` prenant en paramètre un entier `nb` et retournant la valeur `nb! = nb * (nb-1) * ... * 1`. Il faut donc tester le comportement de la fonction pour différents nombres entiers :

— un nombre positif, par exemple 6, pour vérifier que la fonction calcule correctement la factorielle d'un nombre,
— la valeur 0, pour vérifier que le cas particulier du 0 est traité correctement,
— un nombre négatif pour vérifier que la fonction sait traiter les appels avec des valeurs de paramètres incorrectes (on supposera que la fonction retourne dans ce cas -1),
— la valeur 1 qui correspond au cas limite : c'est le premier entier qui est traité comme un nombre positif (ni un cas particulier, ni une erreur).

Il n'est pas contre pas nécessaire de tester la fonction avec les valeurs 2, 3, 4, 5 et 6 car la fonction traite ces nombres de manière similaire. Tester 6 est suffisant.

Remarque : Comme `nb!` augmente très vite, on arrivera assez vite à un dépassement de capacité. (Par exemple, 20 ! est égal à 2432902008176640000 et cette valeur ne peut pas être stockée dans un `int`). Il faudrait tester le comportement de la fonction lorsqu'on l'appelle avec une "grande" valeur (30 par exemple).

Le test de la fonction `factorielle` pourrait être :

```
void test_factorielle()
{
  if(factorielle(1) != 1)
    affichage("Pb avec 1");

  if(factorielle(0) != 1)
    affichage("Pb avec 0");

  if(factorielle(-5) != -1)
    affichage("Pb avec -5");

  if(factorielle(6) != 720)
    affichage("Pb avec 10");

  if(factorielle(10) != 3628800)
    affichage("Pb avec 10");
}
```

Dans cette fonction, on teste si la valeur retournée par la fonction est égale au résultat attendu. Si ce n'est pas le cas, on affiche un message d'erreur. Cette fonction pourrait être améliorée : on pourrait afficher le nombre de tests qui ne sont pas corrects, ajouter un message signalant quand tous les tests se sont passés sans erreur, etc.

Attention : Si la fonction retourne une valeur flottante (`double`), le test doit être différent. En effet, on ne peut pas comparer deux valeurs flottantes avec l'opérateur `==` à cause des problèmes de précision dans la représentation des nombres flottants. Il faut comparer si la valeur absolue de la différence entre les deux nombres est inférieur à un seuil (très petit). Par exemple, si `attendu` et `calcule` sont des variables contenant respectivement le résultat attendu et le résultat retourné par la fonction, il faut alors faire :

```
if( abs(attendu - calcule) <= 0.0001)
  affichage("Le test est bon");
else
  affichage("Problème dans le test");
```

Remarque : Il est possible d'écrire d'abord la fonction de tests avant d'implémenter la fonction pour déterminer les différents comportements de cette fonction suivant les valeurs d'entrée.

Remarque : On ne peut pas tester de cette manière des fonctions de saisie ou d'affichage.

▶ Sur ce thème : **Exercices 5 et 6 du TD 5**

TD5 : Fonctions II (Corrigé)

✓ Exercice 1 : Valeur de retour*

Question 1.1 : À l'exécution de ce code, combien de fois est appelée `f` ? Combien de fois est affiché `hello` ?

```
void f(int a)
{
  if (a<0)
    return;
  affichage("hello\n");
}

int main()
{
  f(3);
  f(-2);
  return 0;
}
```

Correction :

La fonction `f` est appelée 2 fois et `hello` est affichée une seule fois. ◇

Question 1.2 : Ici, quel est l'affichage produit ? Quelles sont les valeurs des variables ?

```
int f(int x)
{
  return 2 * x + 3;
}

int main()
{
  int res = f(3);
  affichage(res, '\n');
  int z = 2;
  res = f(z);
  affichage(res, '\n');
  affichage(f(res), '\n');
  res = f(res);
  affichage(res, '\n');
  affichage(f(res), '\n');
  return 0;
}
```

Correction :

```
9
7
17
17
37
```

La variable `z` a la valeur 2, `res` a la valeur 17. ◇

Question 1.3 : Considérons les fonctions `somme1` et `somme2` définies par :

```
void somme1(int a, int b)
{
  affichage(a+b, '\n');
}

int somme2(int a, int b)
{
  return a+b;
}
```

1. Quelle est la différence entre `somme1` et `somme2` ?
2. Utiliser la fonction `somme1` pour afficher le résultat de $2 + 7$ (sans utiliser l'opérateur `+`). Même question avec la fonction `somme2`.
3. Utiliser la fonction `somme1` pour afficher le résultat de $2 + 7 + 18$ (sans utiliser l'opérateur `+`). Même question avec la fonction `somme2`. Quelle est alors la fonction la mieux programmée ?

Correction :

1. `somme1` prend deux paramètres et ne retourne aucune valeur. Elle affiche le résultat de l'opération $+$. `somme2` prend deux paramètres et retourne le résultat de l'opération $+$.
2.
```
somme1(2,7);
affichage(somme2(2,7), '\n');
```
3. Ce n'est pas possible de faire un calcul (autre qu'une addition) avec `somme1`.
   Avec `somme2` :
```
affichage(somme2(somme2(2,7),18), '\n');
```
   On en déduit que la fonction `somme2` est mieux programmée puisqu'elle est plus générique. ◇

® Exercice 2 : Test de parité

Question 2.1 : Écrire la fonction `estPair` qui affiche si un nombre reçu en paramètre est pair ou non.

Correction :

```
void estPair(int nombre)
{
  if (nombre%2)
    affichage("Le nombre ", nombre, " est pair.\n");
  else
    affichage("Le nombre ", nombre, " est impair.\n");
}
```
◇

Question 2.2 : Ré-écrire la fonction `estPair()` pour que cette dernière renvoie `true` si le nombre reçu en paramètre est pair, `false` sinon.

Correction :

```
bool estPair(int nombre)
{
  return nombre%2==0;
}

```

```
int main()
{
  if (estPair(12))
    affichage("12 est pair\n");
  else
    affichage("12 est impair\n");
  return 0;
}
```

◇

✓ Exercice 3 : Moyenne de deux nombres*

— Écrire la fonction qui reçoit deux nombres flottants en argument et qui calcule et renvoie leur moyenne.
— Que se passe-t-il si l'on souhaite faire la moyenne de 2 et de 3 ?

Correction :

```
double moyenne(double a, double b)
{
  return (a+b)/2;
}

int main()
{
  affichage("Moyenne de 2 et 3 = ", moyenne(2,3), '\n');
  return 0;
}
```

Il n'y a aucun problème pour faire la moyenne de deux entiers car comme les paramètres sont spécifiés comme étant de type `double`, les valeurs 2 et 3 sont implicitement convertis en 2.0 et 3.0 respectivement. Dans tous les cas, `(a+b)` est de type `double` et la division n'est pas une division entière (le dénominateur est automatiquement converti en `double`). ◇

✓ Exercice 4 : Année bissextile**

Écrire une fonction qui permet de determiner si une année est bissextile. On rappelle qu'une année est bissextile si

— elle est divisible par 4
— mais n'est pas divible par 100
— sauf si elle est divisible par 400

Ainsi 2008 était bissextile, 1900 n'était pas bissextile et 2000 était bissextile.

Correction :

```
bool estBissextile(int annee)
{
  return (annee%4==0 && (annee%100!=0 || annee%400==0));
}
```

◇

Exercice 5 : Produit d'entiers**

Question 5.1 : Définir la fonction `produit` qui calcule et renvoie le produit des entiers compris entre n_1 et et n_2 inclus. Si $n_1 \leq n_2$, alors ce produit est égal à $n_1 * (n_1 + 1) * ... * n_2$.

Correction :

```
int produit(int n1, int n2)
{
  if (n1>n2)
  {
    int tmp = n1;
    n1 = n2;
    n2 = tmp;
  }
  int resultat = n1;
  while (n2 > n1)
  {
    resultat *= n2;
    n2--;
  }
  return resultat;
}
```

Attention, `resultat` peut très vite dépasser la capacité de stockage des `int`. Il vaut donc mieux déclarer la variable `resultat` et le type de retour de la fonction comme un `unsigned long int` voire comme `double` suivant les valeurs des paramètres que l'on veut tester. ◇

Question 5.2 : Définir une fonction `test_produit` qui teste la fonction de la question précédente.
Correction :

```
void test_produit()
{
  if(produit(2,5) != 120)
    printf("Probleme avec 2 et 5");

  if(produit(5,2) != 120)
    printf("Probleme avec 5 et 2");

  if(produit(5,5) != 5)
    printf("Probleme avec 5 et 5");

  if(produit(-1,1) != 0)
    printf("Probleme avec -1 et 1");
}
```

◇

® Exercice 6 : Série harmonique**

La série harmonique est la série définie pour tout $n > 0$ par H_n :

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

Question 6.1 : Définir la fonction `harmonique` prenant en paramètre un entier n et retournant la valeur H_n (la fonction retournera -1 si H_n n'est pas défini).
Correction :

```
double harmonique(int n)
{
  if(n < 1)
    return -1;

```

```
  double res = 1;
  while(n > 1)
  {
    res += 1./n;
    n--;
  }
  return res;
}
```

Question 6.2 : Définir une fonction `test_harmonique` testant la fonction `harmonique`.
Correction :

```
void test_harmonique()
{
  if( abs(harmonique(5) - 2.2833) > 0.0001)
    affichage("Problème avec H(5)\n");

  if( abs(harmonique(20) - 3.597739) > 0.0001)
    affichage("Problème avec H(20)\n");

  if( abs(harmonique(1) - 1) > 0.0001)
    affichage("Problème avec H(1)\n");

  if( abs(harmonique(0) - -1) > 0.0001)
    affichage("Problème avec H(0)\n");

  if( abs(harmonique(-10) - -1) > 0.0001)
    affichage("Problème avec H(10)\n");
}
```

TP5 : Fonctions II (Corrigé)

Exercice 7 : Définitions de fonctions et appels*

Pour chaque cas, définir la fonction demandée. Donner également un exemple d'appel de la fonction dans un programme principal.

1. Fonction calculant la valeur absolue d'un nombre.

Correction :

```
double absolue(double nb)
{
  double resultat=nb;
  if (nb<0)
    resultat*=-1;
  return resultat;
}

int main()
{
  int nombre;
  saisie(nombre);
  affichage("la valeur absolue de ", nombre, " est ",
           absolue(nombre), '\n');
  return 0;
}
```

◇

2. Fonction demandant la saisie d'un entier positif ou nul et répétant la saisie jusqu'à ce que ce nombre soit correct.

Correction :

```
int saisiePosNul()
{
  int nb =-1;

  while ( nb < 0))
  {
    //saisie controlee d'un nombre positif ou nul
    affichage("Saisir un nombre positif ou nul \n");
    saisie(nb);
  }
  return nb;
}

int main()
{
  int nombre;
  nombre=saisiePosNul();
  affichage(nombre, " est positif ou nul \n");
  return 0;
}
```

◇

3. Fonction permettant la saisie d'un entier compris dans un intervalle $[a, b]$ (Par exemple, saisie d'un entier entre 10 et 20) et permettant d'effectuer des saisies jusqu'à ce que ce nombre soit correct en indiquant à l'utilisateur s'il doit donner un nombre plus petit ou plus grand.

Correction :

```
int saisieBornee(int a, int b)
{
  int nb;
  //saisie controlee avec indications nombre compris entre a et b (inclus)
  affichage("Saisir un nombre entre ", a, " et ", b, " (inclus) \n");
  saisie(nb);
  while ( nb < a || nb > b)
  {
    if (nb<a)
      affichage("Plus grand ! \n");
    else
      affichage("Plus petit ! \n");
    affichage("Saisir un nombre entre ", a, " et ", b, " (inclus) \n");
    saisie(nb);
  }
  return nb;
}

int main()
{
  int nombre;
  nombre=saisieBornee(10,20);
  affichage(nombre, " est dans le bon intervalle \n");
  return 0;
}
```

Exercice 8 : Discriminant d'un polynôme de second degré*

Le but de cet exercice est de créer une fonction `discriminant` permettant de calculer et retourner la valeur du discriminant d'un polynôme de second degré $ax^2 + bx + c = 0$.

Rappel : Le discrimant de l'équation $ax^2 + bx + c = 0$ est égal à $b^2 - 4ac$. Cette équation admet deux solutions réelles si le discriminant est strictement positif, une solution réelle si le discriminant est nul, et pas de solution réelle autrement.

Question 8.1 : Définir la fonction `discriminant`.

Question 8.2 : Écrire un programme affichant le nombre de racines réelles d'une équation $ax^2 + bx + c = 0$ où les coefficients a, b et c sont saisis par l'utilisateur.

Correction :

```
/*
 * Fonction retournant la valeur du discriminant du polynome a*x*x + b *x + c = 0
 */
double discriminantPlySecondDegre (int a, int b, int c)
{
  return b*b-4*a*c;
}

int main()
{
  int a, b, c;
  affichage("Saisir les coefficients a, b et c de l'equation");
  saisie(a);
```

```
  saisie(b);
  saisie(c);

  double discriminant = discriminantPlySecondDegre (a,b,c);

  if (discriminant>0)
    affichage("Deux racines réelles \n");
  else if (discriminant==0)
    affichage("Racine réelle double \n");
  else
    affichage("Aucune solution réelle \n");

  return 0;
}
```

Exercice 9 : Calcul de x^n quand n est un entier positif ou nul*

Question 9.1 : Écrire une fonction calculant x^n quand n est un entier positif. Appeler cette fonction dans le programme principal.

Correction :

```
double puissance(double x, int n)
{
  double resultat=1;
  int compteur =n;
  while (compteur>0)
  {
    resultat *= x;
    compteur--;
  }
  return resultat;
}

int main()
{
  double nb;
  int p;

  affichage("Donner un nombre \n");
  saisie(nb);
  affichage("Donner une puissance \n");
  saisie(p);
  affichage(nb, " puissance ", p, " = ", puissance(nb,p), '\n');
  return 0;
}
```

Question 9.2 : Définir la fonction `test_puissance` qui teste la fonction `puissance`.

Correction :

```
void test_puissance()
{
  if( abs(puissance(2,3) - 8) > 0.001)
    affichage("Probleme avec 2 puissance 3");

  if( abs(puissance(-2,3) - -8) > 0.001)
    affichage("Probleme avec -2 puissance 3");
```

```

  if( abs(puissance(-2,1) - -2) > 0.001)
    affichage("Probleme avec -2 puissance 1");

  if( abs(puissance(-17,0) - 1) > 0.001)
    affichage("Probleme avec -17 puissance 0");

  if( abs(puissance(-2,-1) - -1) > 0.001)
    affichage("Probleme puissances négatives");
}
```

◇

Exercice 10 : Calcul du PGCD**

L'algorithme d'Euclide calculant le plus grand diviseur commun ($PGCD$) de deux nombres entiers strictement positifs peut être décrit comme suit : *"tant que les 2 nombres sont différents soustraire le plus petit du plus grand. Une fois égaux, afficher l'un d'eux : c'est le PGCD."*

Remarque : Le PGCD ne dépend pas du signe des nombres passés en paramètre. Par exemple, $PGCD(-5,-3) = PGCD(5,3)$. De plus, $PGCD(0,x) = PGCD(x,0) = x$ pour tout entier x.

Question 10.1 : Faire tourner cet algorithme sur papier avec les entiers 18 et 14.
Correction :

```
n1 = 18
n2 = 14

n2<n1 donc n1=n1-n2 n1 = 4
n1<n2  donc n2=n2-n1  n2 = 10
n1<n2  donc n2=n2-n1  n2 = 6
n1<n2  donc n2=n2-n1  n2 = 2
n2<n1  donc n1 = n1-n2  n1 = 2
n1 = n2  donc pgcd = 2
```

◇

Question 10.2 : Définir la fonction `pgcd` qui calcule et retourne le *pgcd* de deux arguments entiers.
Correction :

```
int pgcd(int n1, int n2)
{
  if(n1 < 0)
    n1 = -n1;
  else if(n1 == 0)
    return n2;

  if(n2 < 0)
    n2 = -n2;
  else if(n2 == 0)
    return n1;

  while (n1 != n2)
  {
  if (n1>n2)
    n1 = n1-n2;
  else
    n2 = n2-n1;
  }
  return n1;
}
```

◇

Question 10.3 : Définir une fonction `test_pgcd` testant la fonction `pgcd`.
Correction :

```
void test_pgcd()
{
  if(pgcd(8,14) != 2)
    printf("Probleme avec PGCD(8,14)\n");

  if(pgcd(-8,-14) != 2)
    printf("Probleme avec PGCD(-8,-14)\n");

  if(pgcd(8,-14) != 2)
    printf("Probleme avec PGCD(8,-14)\n");

  if(pgcd(25,100) != 25)
    printf("Probleme avec PGCD(25,100)\n");

  if(pgcd(23,14) != 1)
    printf("Probleme avec PGCD(23,1)\n");

  if(pgcd(8,0) != 8)
    printf("Probleme avec PGCD(8,0)\n");

  if(pgcd(0,8) != 8)
    printf("Probleme avec PGCD(0,8)\n");
}
```

◇