

Systeme d'Information

Partie II Programmation en PHP

<http://www.agroparistech.fr/Systeme-d-Information>



Laurent Orseau

UFR d'Informatique

Département MMIP

`laurent.orseau@agroparistech.fr`

AgroParisTech Grignon 1A

Année 2010-2011

Table des matières

1	Introduction au PHP	6
2	Programmation en PHP	8
2.1	Bases de la programmation	8
2.2	Premier programme	9
2.2.1	Notion d'exécution	10
2.3	Variables	11
2.3.1	Types de variables	11
2.3.2	Affectation de valeur à une variable	11
2.3.3	Utilisation de la valeur d'une variable	12
2.3.4	Autres opérateurs d'affectations	14
2.3.5	Conventions et nom de variables	15
2.4	Notion de bloc d'instructions	15
2.5	Structures de contrôle	16
2.6	La condition if	16
2.6.1	Utilisation du else	16
2.6.2	Imbrication des conditions	17
2.7	Les fonctions	20
2.7.1	Définition de fonction	20
2.7.2	Utilisation d'une fonction	21
2.7.3	Valeur de retour par return et affichage à l'écran	23
2.7.4	Portée des variables	24
2.7.5	Composition de fonctions	25
2.7.6	L'abstraction : Ne jamais écrire deux fois la même chose	26
2.7.7	Méthode diviser pour résoudre	29
2.7.8	Fonctions, Abstraction et Diviser pour résoudre : Conclusion	34
2.8	La boucle while	35
2.8.1	Imbrications	37
2.9	La boucle do...while	37
2.10	La boucle for	38
2.11	Tableaux	40
2.11.1	Parcours en boucle	41
2.11.2	La boucle foreach	41

2.11.3	Tableaux associatifs	42
3	Le paquetage AgroSIXPack	44
3.1	Introduction	44
3.2	Fonction principale	44
3.3	Affichage	45
3.4	Formulaires : Interaction avec l'utilisateur	47
3.4.1	Création et affichage de formulaires	50
3.4.2	Traitement des données de formulaire	53
3.5	Sessions	55
3.6	Modification du style	56
3.6.1	Style agro par défaut	57
4	PHP/MySQL	58
4.1	Connexion à la base de données	58
4.1.1	Exécution de requêtes SQL	59
4.1.2	Requêtes de type SELECT	59
4.1.3	Autres types requêtes	60
4.1.4	Échappement des apostrophes	61
4.2	Fermeture de la connexion à la base	61
4.3	Exemple complet	62
4.4	Pour aller plus loin	63
5	Travaux Dirigés PHP	64
TD 1	: Variables et fonctions	65
Exercice 1	: Premiers pas	65
Exercice 2	: Variables	66
Exercice 3	: Correction d'erreurs	66
Exercice 4	: Fonction de concaténation de chaînes de caractères	66
Exercice 5	: Abstraction	67
Exercice 6	: Variables locales	67
Exercice 7	: Multilinguisme	68
TD 2	: Boucles et tableaux	70
Exercice 1	: 99 Bouteilles de bière	70
Exercice 2	: Triangle	70
Exercice 3	: Jeux de mots	71
Exercice 4	: Vente en ligne	73
TD 3	: Tableaux et formulaires	76
Exercice 1	: Formulaires	76
Exercice 2	: Formulaire auto-validé	78
TD 4	: Sessions et tableaux associatifs	80
Exercice 1	: Compteur	80
Exercice 2	: Labyrinthe	80

TD 5 : PHP/MySQL	84
Exercice 1 : Les Simpson	84
TD 6 : PHP/MySQL, Modélisation	85
Exercice 1 : Réseau social	85
A PHP : Addendum	86
A.1 Signatures de fonctions	86
A.2 Quelques fonctions et structures supplémentaires	87
A.2.1 print_r	87
A.2.2 rand	88
A.2.3 include	88
A.2.4 break	88
A.2.5 isset	88
A.2.6 array_key_exists	88
A.2.7 unset	89
A.3 Chaînes de caractères	89
A.3.1 Comparaison de chaînes de caractères	89
A.3.2 Guillemets simples et doubles, échappement de guillemet	89
A.4 Variables globales	90
A.5 Gestion des erreurs	91
B Indentation en PHP	93
C Ressources et aide en ligne	95
D Utiliser Notepad++, PHP et MySQL chez vous	96
D.1 Éditeur de texte : Notepad++	96
D.2 Navigateur : Firefox	96
D.3 Serveur Apache/Base de données : EasyPHP	96
D.3.1 Configuration d'EasyPHP	97
D.4 Paquetage AgroSIXPack	97

Chapitre 1

Introduction au PHP

Que ce soit dans un régulateur de chauffage, dans un téléphone portable, dans une montre, dans une voiture, ou bien sûr dans un ordinateur, dès qu'il y a un peu d'électronique, il y a de la programmation. Tout logiciel, que ce soient des tableurs, des jeux, des logiciels de gestion, ou même le système d'exploitation des ordinateurs, a été programmé par des ingénieurs et des techniciens. De nombreux scientifiques voient même le cerveau humain comme un ordinateur (certes assez différent de ce à quoi nous sommes habitués), et la manière dont il fonctionne comme un gigantesque programme. D'ailleurs, des études sont faites pour tenter de simuler des parties du cerveau, ce qui nécessite de bonnes connaissances en programmation.

La plupart du temps, les logiciels sont suffisamment bien faits pour que l'on n'ait pas à savoir programmer. Mais comprendre la manière dont un ordinateur fonctionne est un atout important pour n'importe quel utilisateur. Connaître la programmation est aujourd'hui une nécessité, surtout pour des ingénieurs.

Un langage de programmation est un langage, similairement au langage humain, permettant de transmettre des informations ou, en l'occurrence, des instructions de l'homme à la machine. Ces langages existent pour faciliter le travail du programmeur, en lui permettant de dialoguer dans un langage que les 2 parties comprennent.

En 1994, Rasmus Lerdorf a créé le langage PHP (pour "PHP : Hypertext Preprocessor") pour gérer son propre site Web. PHP s'avéra fort utile et de nombreuses personnes commencèrent à l'utiliser. Il existe d'autres langages pour la création de sites Web dynamiques, comme Java, l'ASP, etc. Les avantages du PHP sont qu'il est libre¹ et gratuit, aujourd'hui très répandu, et assez facile d'utilisation tout en restant relativement complet.

Le PHP est un langage de programmation au même titre que Matlab ou C. Sa particularité est d'être très pratique pour créer des sites Web ; cependant nous allons l'utiliser comme simple outil de programmation, l'aspect "site Web" ne nous intéressant que pour obtenir un rendu visuel.

Nous commencerons par voir les bases de la programmation en PHP, à travers les variables, les fonctions, les boucles et les tableaux. Ensuite nous fournissons le paquetage

1. Ce qui signifie principalement qu'on a le droit de le modifier et de le redistribuer librement, mais attention cependant à bien lire les termes de la licence.

AgroSIXPack qui permet de s'abstraire de l'aspect technique et peu intéressant dans ce cours de la création de site Web. Un certain nombre de fonctions y sont définies pour vous permettre de vous focaliser sur l'aspect programmation. Enfin, nous utiliserons PHP avec MySQL pour effectuer des requêtes sur une base de données.

Chapitre 2

Programmation en PHP

PHP est avant tout un langage de programmation à part entière, et il peut tout à fait être utilisé dans un contexte autre que celui des sites Web ; c'est d'ailleurs de cette manière que nous allons l'utiliser.

Commençons par donner le leitmotiv de la programmation :

Ne jamais écrire deux fois la même chose.

Cela implique qu'il faut chercher à tout automatiser, et éviter au maximum le copier/coller. Pourquoi ? Parce que le copier/coller est une des causes d'erreurs les plus fréquentes en programmation, car on oublie facilement de modifier quelque chose qui aurait dû l'être et, surtout, cela empêche de rendre les programmes clairs, concis, structurés et *généraux*.

Tout automatiser signifie que l'on fait faire à l'ordinateur toutes les tâches répétitives, pendant que le programmeur peut se concentrer sur l'essentiel : ajouter de nouvelles fonctionnalités au produit actuel.

C'est là le principe général de la programmation : dire à l'ordinateur ce qu'il faut faire, ainsi que d'éviter au maximum de faire plusieurs fois "à la main" toute tâche répétitive. Il fera tout ce qu'on lui demande, à condition de le lui demander de la bonne manière. Il est très serviable, mais comme personne n'est parfait, il est un peu pointilleux sur les détails. Heureusement, avec une once de rigueur, on apprend vite comment dialoguer avec lui.

2.1 Bases de la programmation

La plupart des langages de programmation possèdent les éléments suivants :

- des *variables*, permettant de stocker des valeurs,
- des *tableaux* ou des listes, qui sont une succession de variables que l'on peut parcourir automatiquement,
- des instructions de base (des *primitives*), qui permettent d'afficher des valeurs à l'écran, lire/écrire dans des fichiers, faire des calculs arithmétiques, etc.,

- des *branchements conditionnels*, permettant d'effectuer certaines instructions selon la valeur de certaines variables,
- des *boucles* d'itération, permettant d'effectuer un même bloc d'instructions plusieurs fois de suite, jusqu'à ce qu'une condition d'arrêt soit vérifiée,
- des *fonctions*, qui sont des suites d'instructions *paramétrées* permettant entre autres d'éviter d'écrire plusieurs fois les mêmes choses.

PHP possède ces éléments et nous les détaillerons un par un dans la suite.

Il faut cependant bien retenir que l'élément considéré le plus important aujourd'hui est la *fonction*, car elle permet d'avoir une programmation *structurée*, permettant une modification, une extension, et un maintien facile du programme. Nous verrons comment utiliser les fonctions dans la suite.

2.2 Premier programme

Le *code* d'un programme est un texte lisible à la fois par un humain et par l'ordinateur. Il s'agit d'une *suite d'instructions* que l'ordinateur *exécutera* de la même manière qu'une recette de cuisine, c'est-à-dire en lisant les lignes une par une, dans l'ordre, et en exécutant une action (ou plusieurs) à chaque ligne.

Voici un tout premier programme PHP :

```
1 <?php
2 function main() {
3     // Message de bienvenue :
4     printline('Bonjour le monde !');
5 }
6 ?>
```

Ce programme, une fois exécuté sur l'ordinateur, affiche à l'écran :

```
Bonjour le monde !
```

Remarquons plusieurs parties : `<?php` apparaît toujours en début de programme et `?>` toujours à la fin. Ces "balises" signifient qu'à l'intérieur il y a du code PHP. Si elles ne sont pas présentes, le texte n'est pas considéré comme du PHP et ne peut donc pas être exécuté.

Nous *déclarons* la fonction `main` dont la *portée* est définie par les accolades. Il est obligatoire de déclarer cette fonction `main`. Dans la suite de ce chapitre, nous l'omettrons la plupart du temps pour nous concentrer sur l'important, sauf dans les exemples de programmes entiers ; de même, nous omettrons la plupart du temps les `<?php ?>`. À noter que l'obligation de déclarer une fonction `main` est due au paquetage `AgroSIXPack`, que nous détaillerons dans un prochain chapitre.

À l'intérieur de la fonction `main`, une instruction fait *appel* à la fonction prédéfinie (la *primitive*)¹ `printline`, qui affiche à l'écran ce qui lui est donné en paramètres, dans les

1. En réalité, `printline` n'est pas une primitive car elle est définie à partir de la fonction primitive

parenthèses, puis passe une ligne. Le texte à afficher à l'écran est écrit dans le programme avec des guillemets car ce n'est pas du code ; cela permet à l'ordinateur de savoir qu'il ne faut pas chercher à *interpréter* ce qui est entre guillemets comme des instructions, mais de le considérer comme du texte simple, sans signification particulière pour le programme.

À la fin de l'instruction, il y a un point virgule ";" pour signaler à l'ordinateur que l'instruction courante s'arrête là ; c'est un point de repère permettant de séparer les instructions.

Le texte commençant par `//` est un commentaire, il n'est donc pas interprété par PHP, qui passe directement à la suite. On peut mettre plusieurs lignes de commentaires entre `/*` et `*/` :

```
1 // Commentaire sur une ligne
2
3 /*
4 Commentaire
5 sur plusieurs
6 lignes
7 */
```

2.2.1 Notion d'exécution

L'ordinateur, lorsqu'il reçoit un fichier PHP à exécuter, regarde chaque ligne l'une après l'autre. Chaque ligne de code comprend une ou plusieurs *instructions* que l'ordinateur exécute dans l'ordre.

Le programme suivant :

```
1 <?php
2 function main() {
3     printline('Bonjour le monde !');
4     printline('Au revoir.');
```

affichera toujours :

```
Bonjour le monde !
Au revoir.
```

et jamais :

```
Au revoir.
Bonjour le monde !
```

Par la suite, nous suivrons le cours de l'exécution d'un programme PHP à l'aide d'un *pointeur d'instruction*. Ici, le pointeur d'instruction est tout d'abord positionné sur l'`print`, mais de votre point de vue la différence n'a pas d'importance.

truction de la ligne 2, laquelle est exécutée. Le pointeur se déplace ensuite sur l’instruction suivante, à la ligne 3, qui est alors aussi exécutée, et ainsi de suite dans l’ordre jusqu’à la fin du programme.

Après ce rapide exemple, nous expliquons maintenant le fonctionnement des variables, avant de revenir sur celui des fonctions.

2.3 Variables

Les variables sont des cases mémoires de l’ordinateur dans lesquelles sont stockées des valeurs. Il est possible de lire le contenu d’une telle case ou de le modifier.

Ces valeurs peuvent donc changer au cours du temps (au cours des exécutions des instructions), et le comportement du programme dépendra de ces variables.

En PHP, une variable est identifiée par le symbole `$` suivi d’une chaîne de caractères représentant son nom, par exemple `$couleur`.

2.3.1 Types de variables

Les différentes sortes de valeurs que peuvent contenir les variables sont appelées des *types*. Il y a notamment le type *chaîne de caractères*, le type *nombre* (avec les sous-types *entier*, *réel*, etc.), le type *booléen* abrégé en *bool* (soit `true` soit `false`)², le type composé *array*, le type *resource*, etc.

Le type d’une variable définit l’ensemble des opérations que l’on peut effectuer sur la variable. Par exemple, il n’est pas possible de multiplier 2 chaînes de caractères.

PHP est un langage à *typage faible*, c’est-à-dire que l’on n’écrit pas explicitement le type de la variable que l’on utilise.

2.3.2 Affectation de valeur à une variable

Pour modifier la valeur de la case mémoire d’une variable, on utilise un *affectation* avec le signe `=`.

À gauche du signe `=`, on place la variable dont on veut modifier la valeur. À droite du signe `=`, on place la valeur :

```
1 $x = 42;
```

La valeur de `$x` ne sera modifiée qu’une fois que le programme aura exécuté cette instruction, c’est-à-dire qu’une fois que le pointeur d’instruction sera passé sur cette ligne.

Attention ! Il ne s’agit pas du signe `=` mathématique, mais bien d’une instruction de *modification d’état*. Il serait plus correct de lire `←` à la place du signe `=` :

2. D’après George Boole, qui inventa l’algèbre de Boole au XIX^e siècle, permettant de faire des calculs à partir des seules valeurs `true` et `false`.

```
1 $x ← 42;
```

ce qui signifierait "faire l'action de mettre la valeur 42 dans la case mémoire de la variable `$x`" ou plus simplement "`$x` reçoit la valeur 42". Cette opération s'effectue toujours de la droite vers la gauche, jamais dans l'autre sens.

Pour afficher la valeur que contient la variable, il suffit de faire :

```
1 $x = 42;  
2 printline($x);
```

Ici, il n'y a pas de guillemet autour de `$x` dans l'appel au `printline`, car PHP doit d'abord transformer la variable `$x` en son contenu, ce qui revient à transformer l'instruction en :

```
1 printline(42);
```

Par exemple, dans le code :

```
1 $x = 42;  
2 printline($x);  
3 newline();  
4  
5 $x = 17;  
6 printline($x);  
7 newline();  
8  
9 $x = 23;  
10 printline($x);
```

les instructions sont exécutées dans l'ordre donc l'ordinateur affiche successivement les valeurs 42, puis 17, puis 23. À la fin de l'exécution du programme, `$x` a donc la valeur 23.

On peut aussi mettre (stocker) des *chaînes de caractères*, c'est-à-dire du texte, dans une variable :

```
1 $nom = 'Philip Kindred Dick';  
2 printline($nom);
```

2.3.3 Utilisation de la valeur d'une variable

Il est possible d'utiliser la valeur d'une variable dans un calcul :

```
1 $x = 23;  
2 $n = 42;  
3 $y = $n*5+3+$x;
```

Un calcul peut être utilisé en partie droite du = ou dans le `printline` :

```
1 $x = 23;  
2 $n = 42;  
3 $longueur = 17;  
4 $n = $x*2+$n;  
5 printline(3*$x + $longueur);
```

La partie droite du = s'appelle une *expression*. Une expression est une partie d'une instruction où l'ordinateur peut avoir besoin d'effectuer plusieurs calculs avant d'arriver au résultat sous forme d'une valeur simple (un nombre, une chaîne de caractères, etc.). Un expression est toujours un calcul, dans le sens où il y a transformation de valeurs (numériques, textuelles ou autres) en d'autres valeurs. Au final, le résultat d'une expression, pour qu'il soit utile à quelque chose, doit être utilisé dans une instruction (comme `printline` ou une affectation).

Par exemple, l'instruction suivante, seule, est inutile :

```
1 $x+2*$y;
```

car le résultat du calcul n'est utilisé d'aucune façon. C'est un calcul "dans le vide". S'il s'était trouvé en partie droite d'une affectation par exemple, il aurait été utile.

Une expression peut être aussi complexe qu'on le veut, mais il est préférable de couper les grands calculs en plusieurs petits.

Exercice 1 *Que fera le programme suivant :*

```
1 <?php  
2 function main() {  
3     $i = 5;  
4     printline($i);  
5  
6     $i = i+1;  
7     printline($i);  
8  
9     $i = i+10;  
10    printline($i);  
11 }  
12 ?>
```

2.3.4 Autres opérateurs d'affectations

L'opérateur = est suffisant pour tout type d'affectation, mais il existe d'autres opérateurs qui permettent d'écrire plus simplement certaines affectations courantes.

Voici quelques-uns des plus utilisés :

Opérateur	Signification
<code>\$x++</code>	<code>\$x = \$x + 1</code>
<code>\$x--</code>	<code>\$x = \$x - 1</code>
<code>\$x += A</code>	<code>\$x = \$x + A</code>
<code>\$x -= A</code>	<code>\$x = \$x - A</code>
<code>\$s .= C</code>	<code>\$s = \$s . C</code>

Par exemple :

```
1 $x = 5;
2 $x++;
3 printline($x);
```

affichera 6.

L'opérateur "." est l'opérateur de *concaténation* des chaînes de caractères. Il permet de "coller" deux chaînes de caractères l'une à la suite de l'autre :

```
1 $x = 42;
2 $x += 11;
3 printline('La val' . 'eur de x est ' . $x . ' !');
```

affichera :

```
La valeur de x est 53 !
```

où `$x` a d'abord été transformé automatiquement de nombre en chaîne de caractères.

De même :

```
1 $nom = 'Jean';
2 $nom .= '-Sébastien';
3 $nom .= ' Bach';
4 printline('Votre nom : ' . $nom);
```

affichera :

```
Votre nom : Jean-Sébastien Bach
```

2.3.5 Conventions et nom de variables

Il est conseillé de n'utiliser que des caractères alphabétiques (pas de symbole particulier comme + ou @, etc.) dans les noms des variables. Les caractères accentués ne sont pas recommandés. Le symbole _ peut cependant être utilisé pour séparer les mots lorsque la variable possède un nom composé : `$vitesse_electron`. Les chiffres peuvent apparaître, mais jamais en début de nom.

Il est par ailleurs vivement conseillé d'utiliser des noms de variables explicites (par exemple `$hauteur` ou `$couleur_yeux`) plutôt que des noms abscons comme `$toto` ou `$a`.

2.4 Notion de bloc d'instructions

Nous allons souvent avoir besoin de regrouper une suite donnée d'instructions. C'est ce que l'on appelle un *bloc d'instructions*, et il commence toujours par une accolade ouvrante { et finit toujours par une accolade fermante } :

```
1 {
2   instruction1
3   instruction2
4   instruction3
5   ...
6 }
7
```

Voici un exemple de bloc d'instructions :

```
1 {
2   $taille = 79;
3   printline('la taille est ' . $taille);
4   $forme = 'rectangulaire';
5   printline('la forme est' . $forme);
6 }
7
```

Notez le passage à la ligne après la première accolade ainsi qu'avant l'accolade fermante. Notez aussi que tout ce qui se trouve à l'intérieur du bloc d'instructions a été décalé vers la droite d'un nombre fixe d'espaces (4 semble une valeur convenable). Comme nous le verrons par la suite, une fois que l'on a fermé le bloc d'instructions, on revient au même niveau que les accolades.

Cela s'appelle *l'indentation*. On dit que les lignes ont été indentées, ou *tabulées*. L'indentation est très importante pour rendre un programme lisible par l'homme (généralement la machine n'en a pas besoin). Lire la section B pour plus de détails sur la nécessité de l'indentation.

Les blocs d'instructions sont nécessaires aux *structures de contrôle*.

2.5 Structures de contrôle

Les structures de contrôle sont des éléments de base des langages de programmation qui permettent de modifier le comportement du programme selon la valeur des variables.

Normalement, les instructions sont exécutées les unes à la suite des autres, dans l'ordre. Les structures de contrôle (de l'ordre d'exécution des instructions) permettent de modifier cet ordre au moment de l'exécution du programme.

Elles sont de trois sortes :

- les conditions,
- les boucles,
- les fonctions.

2.6 La condition `if`

La condition³ `if` permet d'exécuter une suite d'instructions uniquement si un *test* donné est vérifié. Par exemple :

```
1 if($x > 5) {  
2     printline('x est supérieur à 5');  
3 }
```

Ce qui se trouve entre les *parenthèses* qui suivent le `if` s'appelle le *test* de la condition. Un test est une expression dont le calcul doit arriver soit à la valeur `true`, soit à la valeur `false`. Tout ce qui se trouve dans les accolades suivant le `if` sera traité uniquement si le test de la condition est vérifié (sa valeur calculée vaut `true`).

Notez la tabulation ajoutée sur la ligne suivant le `if`.

Les instructions du bloc d'instructions suivant le `if` seront soit toutes exécutées, soit aucune.

2.6.1 Utilisation du `else`

Il est possible d'exécuter des instructions dans le cas où le test du `if` a échoué (le test vaut `false`). Il faut pour cela utiliser le mot-clef `else` :

```
1 $x = 3;  
2 if($x > 5) {  
3     printline('x est supérieur à 5');  
4 } else {  
5     printline('x est inférieur ou égal à 5');  
6 }
```

Notez que le `else` se trouve entre l'accolade fermante des instructions du cas où le test est `true`, et l'accolade ouvrante du cas `false`.

3. Le `if` n'est en aucun cas une boucle !

2.6.2 Imbrication des conditions

Il est tout à fait possible d'imbriquer les conditions :

```

1 $max = 0;
2 if($x > $y) {
3     printline('x est supérieur à y');
4     if($x > $z) {
5         printline('x est la plus grande valeur des trois');
6         $max = $x;
7     } else {
8         printline('z est la plus grande valeur des trois');
9         $max = $z;
10    }
11 } else {
12     printline('y est supérieur à x');
13     if($y > $z) {
14         printline('y est la plus grande valeur des trois');
15         $max = $y;
16     } else {
17         printline('z est la plus grande valeur des trois');
18         $max = $z;
19     }
20 }

```

mais en gardant à l'esprit que le test `if($x > $z)` (par exemple) ne se fera que si la condition du `if($x > $y)` est vérifiée.

Lorsque beaucoup de `if` et de `else` s'enchaînent, il est préférable d'utiliser des `else if`, (ou `elseif`) successifs plutôt que de multiplier les imbrications de blocs d'instructions. C'est-à-dire, au lieu de :

```

1 if($x < 0) {
2     printline('x est négatif');
3 } else {
4     if($x < 10) {
5         printline('x est inférieur à 10');
6     } else {
7         if($x < 100) {
8             printline('x est inférieur à 100');
9         } else {
10            printline('x est supérieur ou égal à 100');
11        }
12    }
13 }

```

il est préférable d'écrire :

```

1 if($x < 0) {
2     printline('x est négatif');
3 } else if($x < 10) {
4     printline('x est inférieur à 10');
5 } else if($x < 100) {
6     printline('x est inférieur à 100');
7 } else {
8     printline('x est supérieur ou égal à 100');
9 }

```

La forme générale du **if** est donc la suivante :

```

1 if( test ) {
2     instruction
3     instruction
4     ...
5 } else {
6     instruction
7     instruction
8     ...
9 }

```

où les instructions peuvent contenir elles aussi des **if**.

Le *test* est une expression (un calcul) qui, une fois évaluée (calculée), doit avoir la valeur **true** ou **false**⁴.

Pour effectuer le test du **if**, on utilise des *opérateurs de comparaison*. Voici les plus courants :

Opérateur	Signification
$A == B$	vaut true ssi l'expression A a la même valeur calculée que B
$A != B$	true ssi A est différent de B
$A < B$	true ssi A est inférieur à B
$A > B$	true ssi A est supérieur à B
$A <= B$	true ssi A est inférieur ou égal à B
$A >= B$	true ssi A est supérieur ou égal à B

Par exemple :

```

1 if($x == 5) {
2     printline('x est égal à 5');
3 }

```

4. En PHP, la valeur **false** est équivalente à la chaîne de caractères vide "" ou à la valeur `null` ou à la valeur 0; toutes les autres valeurs sont équivalentes à **true**.

Note : Les opérateurs `==` et `!=` peuvent aussi être utilisés sur les chaînes de caractères.

Ces opérateurs permettent aussi de comparer des expressions et non seulement des variables ou des valeurs simples :

```
1 if($x+3 == $y*$z) {
2   printline('Ok');
3 }
```

Dans un telle expression, on peut combiner plusieurs comparaisons en une seule en utilisant des *opérateurs logiques* :

Opérateur	Signification
$A \text{ and } B$	vaut true si à la fois A et B ont la valeur true
$A \text{ or } B$	vaut true si A ou B ou les deux ont la valeur true
$A \text{ xor } B$	vaut true si A ou B ont la valeur true , mais pas les deux à la fois
$A \ \&\& \ B$	comme and
$A \ \ B$	comme or
$!A$	non(A) : vaut true si A est false et réciproquement

L'utilisation de `&&` et `||` est déconseillée car moins facilement lisible, bien que courante.

Pour éviter tout problème de priorité entre les opérateurs, c'est-à-dire quels opérateurs agissent en premier, il est conseillé d'entourer les expressions avec des parenthèses :

Par exemple :

```
1 if( ( ($x > 100) and ($x < 200) ) or ($x > 500) ) {
2   printline('100 < x < 200 ou bien x > 500');
3 }
```

2.7 Les fonctions

Une *fonction* est essentiellement un bloc d'instructions auquel on a donné un nom, qui dépend de *paramètres d'entrée*, et qui renvoie une *valeur de sortie*.

Elle permet d'effectuer un calcul ou une tâche intermédiaire de manière à peu près indépendante du reste du programme.

Les intérêts des fonctions sont multiples :

- faciliter la conception (la phase avant l'écriture) du programme : on peut diviser le problème en sous-problèmes, et chacun sera résolu par une fonction, c'est ce que l'on appelle "diviser pour résoudre"⁵,
- rendre le programme plus compact : on peut utiliser plusieurs fois une même fonction pour effectuer des calculs similaires, au lieu de recopier le même code à différents endroits,
- faciliter l'écriture du programme, pour les mêmes raisons,
- rendre le programme plus lisible : il est bien plus pratique de lire `avancer_pion(case_source, case_destination)` qu'un long morceau de code faisant la même chose,
- faciliter la modification d'un programme : on ne modifie qu'une seule fois la fonction alors que si le code se répète, il faut le modifier dans chacune de ses répétitions.

2.7.1 Définition de fonction

Avant de pouvoir être utilisée dans le programme, il faut donner une *définition* de la fonction, c'est-à-dire que l'on doit dire au programme ce qu'elle fait, en "fonction" de quoi.

Voici un exemple de fonction qui fait simplement la somme de 3 valeurs :

```
1 function ajouter3($x, $y, $z) {
2     $somme = $x + $y + $z;
3     return $somme;
4 }
```

Anatomie d'une fonction :

```
1 function nom(variable1, variable2...) {
2     corps
3     ...
4     return expression;
5 }
```

Une fonction commence toujours par le mot clef `function`, suivi du nom qu'on lui donne, suivi d'une liste de variables nommées *arguments* ou *paramètres d'entrée*. Cette liste peut être vide.

5. Ou "diviser pour régner".

Vient ensuite le *corps* de la fonction, qui correspond à la suite d'instructions exécutée par la fonction lorsqu'elle est appelée. Une fonction effectue une transformation des valeurs d'entrée en une valeur de sortie. C'est pourquoi elle se termine par l'instruction `return expression;`, qui a pour objectif de renvoyer en sortie la valeur calculée de l'expression. Cette valeur calculée pourra ainsi être réutilisée dans un autre calcul en dehors de la fonction.

Remarquez que le corps de fonction et le `return` sont nécessairement entourés des accolades de bloc d'instructions.

Une fonction doit toujours elle-même être définie en dehors de tout bloc d'instructions et de toute structure de contrôle.

Contrairement aux autres parties du programme, lorsque PHP "lit" la définition de la fonction, il *n'exécute pas* les instructions qui se trouvent dans le corps de la fonction. À la place, il les garde quelque part dans sa mémoire et les exécutera réellement que lors de **l'appel de fonction**.

2.7.2 Utilisation d'une fonction

Une fois que la fonction est définie, on peut l'utiliser en l'appelant dans le reste du programme :

```
1 <?php
2 function ajouter3($x, $y, $z) {
3     $somme = $x + $y + $z;
4     return $somme;
5 }
6
7 function main() {
8     $n = 5;
9     printline('La valeur de n est ' . $n);
10
11     $n = ajouter3($n, 8, 27);
12     printline('La valeur de n est ' . $n);
13
14     $n = ajouter3(1, 0, 8);
15     printline('La valeur de n est ' . $n);
16 }
17 ?>
```

Ce programme affichera :

```
La valeur de n est 5
La valeur de n est 40
La valeur de n est 9
```

Lors de l'exécution du programme, le pointeur d'instruction parcourt la définition de la fonction (ligne 2) jusqu'à arriver à la fin de son corps. *Ses instructions ne sont pas exé-*

cutées immédiatement, mais elles sont conservées en mémoire. Il en va de même pour la fonction spéciale `main`, mais cette dernière est *exécutée* juste après la lecture complète du fichier PHP, de sorte que le pointeur d'instruction arrive ensuite sur la première affectation (ligne 8).

Ceci affiche donc son résultat et le pointeur d'instruction arrive sur la seconde affectation (ligne 11). Rappelons que l'affectation possède toujours en partie droite une *expression*. Ici, l'expression est un calcul effectué par un *appel de fonction* : on donne 3 valeurs à cette fonction et on récupère le résultat :

```
11 $n = ajouter3($n, 8, 27);
```

Attention, ici, il ne faut pas considérer que c'est la variable `$n` de droite elle-même qui sera envoyée à la fonction, mais uniquement sa **valeur**. Le `$n` de droite est donc transformé en sa valeur **avant** l'appel de fonction, comme si l'on avait :

```
11 $n = ajouter3(5, 8, 27);
```

La variable `$n` de droite n'apparaîtra donc pas dans l'exécution de la fonction, et seule sa valeur y sera. Ceci signifie que la fonction ne peut en aucun cas modifier le contenu de `$n`.

À cause de l'appel de fonction à `ajouter3`, le pointeur d'instruction *saute* alors jusqu'à la ligne 2 qui est le début de la fonction. Cependant, on mémorise quand même le numéro de ligne où s'est effectué le saut, pour pouvoir y revenir ensuite. Dans la fonction, tout se passe comme si on était dans un *petit programme (presque) indépendant* du programme principal : **Les variables de la fonction n'ont aucun rapport avec les variables du programme principal**. On les appelle des *variables locales*. La variable `$x` prend alors la valeur 5, `$y` prend la valeur 8 et `$z` prend la valeur 27. À l'instruction suivante dans la fonction, on calcule la valeur de `$somme`, qui prend donc la valeur 40. On arrive enfin sur l'instruction :

```
4 return $somme;
```

Cette instruction renvoie ("retourne") la valeur contenue dans la variable `$somme` en sortie de fonction. C'est la *valeur de retour*. On sort de la fonction, et les variables locales *perdent leur valeur* : **ces variables n'existent plus en dehors de la fonction**.

Le pointeur d'instruction retourne alors sur la ligne 11 et la valeur de retour remplace l'appel de fonction. L'appel :

```
11 $n = ajouter3(5, 8, 27);
```

s'est donc transformé en, après calcul fait par la fonction :

```
11 $n = 40;
```

L'expression à droite du `=` étant un nombre, elle peut maintenant être affectée à la va-

riable `$n`. Aux instructions suivantes, PHP affiche le résultat, affecte à `$n` une autre valeur calculée par appel de fonction et l'affiche aussi.

2.7.3 Valeur de retour par `return` et affichage à l'écran

Il est important de bien faire la différence entre valeur de retour d'une fonction et affichage d'un résultat à l'écran.

Dans le retour d'une fonction, il n'y a aucune interaction avec l'utilisateur qui regarde l'écran. Tout se passe à l'intérieur du programme, où ce sont les instructions qui "dialoguent", se passent des valeurs et les modifient. Il s'agit uniquement d'un "calcul".

Dans le cas de l'affichage à l'écran de la valeur calculée par la fonction, il y a une interaction directe avec l'utilisateur du programme. C'est-à-dire qu'à chaque fois que cette fonction sera appelée, il y aura un affichage à l'écran. Cela n'est généralement pas souhaitable, car les fonctions peuvent être appelées de nombreuses fois (parfois des millions), ce qui pourrait alors encombrer l'écran d'informations parfaitement inutiles à l'utilisateur.

Reprenons l'exemple du programme précédent : il est possible (et fortement conseillé, pour toutes les raisons indiquées plus haut) de créer une fonction séparée réalisant l'affichage à l'écran de la valeur de `$n`. Cela évitera entre autres d'écrire plusieurs fois la même chose :

```
1 <?php
2 function ajouter3($x, $y, $z) {
3     $somme = $x + $y + $z;
4     return $somme;
5 }
6
7 function afficher_n($val_n) {
8     printline('La valeur de n est ' . $val_n);
9 }
10
11 function main() {
12     $n = 0;
13     afficher_n($n);
14
15     $n = ajouter3(5, 8, 27);
16     afficher_n($n);
17
18     $n = ajouter3(1, 0, 8);
19     afficher_n($n);
20 }
21 ?>
```

Remarquez que la fonction `afficher_n` n'utilise pas le mot clef `return`, car lorsque l'on appelle cette fonction, on n'a pas besoin de récupérer une valeur pour l'utiliser dans un calcul ou la mettre dans une variable. Cette fonction ne fait en effet qu'afficher quelque chose à l'écran. De telles fonctions sans valeur de retour sont parfois appelées *procédures*.

À l'inverse, la fonction `ajouter3` renvoie une valeur mais n'utilise jamais le `printline`.

Il est donc fortement recommandé d'appliquer la règle suivante : *Une fonction qui retourne une valeur ne doit pas utiliser de `printline`.*

Attention, l'instruction `return` doit être la dernière instruction que doit exécuter la fonction ! En effet, dès que le pointeur d'instruction arrive sur le `return`, il sort immédiatement de la fonction avec la valeur de retour. Les instructions suivantes sont alors ignorées.

2.7.4 Portée des variables

Reprenons le code de la fonction `afficher_n` :

```
1 function afficher_n($val_n) {
2     printline('La valeur de n est ' . $val_n);
3 }
```

Notez bien que dans cette fonction, ce n'est pas la variable `$n` que l'on utilise, mais bien l'autre variable `$val_n`, qui reçoit la valeur de `$n` lors de l'appel de fonction. La variable `$n` n'apparaît donc pas dans la fonction `afficher_n`.

Soit le programme suivant :

```
1 <?php
2 function ajouter3($x, $y, $z) {
3     $somme = $x + $y + $z;
4     return $somme;
5 }
6
7 function main() {
8     $somme = 12;
9     $x = 18;
10    printline($somme);
11    printline($x);
12    $n = ajouter3(23, 35, 42);
13    printline($n);
14    printline($somme);
15    printline($x);
16    printline($y);
17 }
18 ?>
```

Une fois le pointeur d'instruction arrivé sur l'instruction de la ligne 8, la variable `$somme` prend la valeur 12, puis la variable `$x` prend la valeur 18. On affiche ensuite leurs valeurs.

À la ligne 12, on appelle la fonction `ajouter3`. Le pointeur d'instruction se déplace alors jusqu'à la ligne 1 qui est le début de la fonction. Ici sont introduites 3 variables d'entrée `$x`, `$y` et `$z`.

Comme nous l'avons dit plus haut, une fonction est un petit programme indépendant du programme principal : les variables sont de *nouvelles* variables et même si elles ont le même nom, ce ne sont pas les mêmes. Ce sont des variables *locales*.

Ceci a pour avantage majeur que l'on peut utiliser les fonctions que d'autres programmeurs ont créé (et cela est extrêmement courant et pratique) sans avoir à connaître les variables que ces fonctions utilisent, ni comment elles sont utilisées. Même si l'on n'utilise que nos propres fonctions, cela permet de grandement faciliter la décomposition du programme en sous-programmes plus simples. Une fois ces fonctions/sous-programmes créés, il n'est plus nécessaire de (entièrement) se souvenir de la manière dont elles ont été définies.

Pour le programme ci-dessus donc, une fois le pointeur d'instruction sur la ligne 2, on sait que les variables `$somme`, `$x`, `$y` et `$z` sont locales à la fonction `ajouter3`. Les valeurs des variables `$somme` et `$x` du programme principal ne sont donc pas modifiées. La fonction calcule ensuite la valeur de `$somme` puis elle retourne la valeur de cette variable. Lorsque l'on sort de la fonction, on retourne directement en ligne 12, pour remplacer l'appel de fonction par sa valeur retournée :

```
12 $n = ajouter3(23, 35, 42);
```

se transforme alors en :

```
12 $n = 100;
```

et `$n` prend donc la valeur 100. Comme on est sorti de la fonction, les variables locales de celle-ci n'existent plus et les variables `$somme` et `$x` ont maintenant les mêmes valeurs qu'avant l'appel de fonction. Ce sont donc ces valeurs qui s'affichent, c'est-à-dire 12 et 18.

Enfin, la dernière ligne cherche à écrire la valeur de la variable `$y`. Mais comme cette variable n'a jamais reçu de valeur (par une affectation), elle n'existe pas dans le programme principal. PHP affiche alors une erreur :

```
Notice: Undefined variable: y in mon_fichier.php on line 16
```

On retrouve d'abord le type de l'erreur, comprenant le nom de la variable non définie, puis le nom de fichier impliqué et le numéro de ligne où a été remarquée l'erreur.

2.7.5 Composition de fonctions

Dans une expression, on peut aussi *composer les appels de fonctions* pour utiliser la valeur de sortie d'une fonction pour la donner en argument en entrée d'une autre fonction. Ceci est comparable à la composition mathématique des fonctions, comme $g(f(5))$.

Par exemple :

```
1 printline(ajouter3(3, 5, multiplier3(8, 6, 4)));
```

Ici, le troisième paramètre de la fonction `ajouter3` est un autre appel à la fonction `multiplier3`, que l'on suppose définie en début de programme. Pour déterminer quelle valeur est envoyée à l'appel de `println`, on commence par calculer la valeur de retour de l'appel qui est le plus profond dans l'imbrication, soit `multiplier3(8, 6, 4)`. Une fois cette valeur calculée, celle-ci remplace l'appel dans l'expression globale :

```
1 println(ajouter3(3, 5, 192));
```

Notez bien qu'il est nécessaire que l'appel à `multiplier3` retourne une valeur numérique, car si elle retournait une chaîne de caractères ou rien du tout, cela n'aurait pas de sens vis-à-vis de `ajouter3` :

```
1 println(ajouter3(3, 5, 'ceci est une chaîne de caractères'));
```

Remarquez qu'il est toujours possible de décomposer les imbrications lors de l'appel en passant par des variables intermédiaires :

```
1 $n1 = multiplier3(8, 6, 4);  
2 $n2 = ajouter3(3, 5, $n1);  
3 println($n2);
```

Ceci est d'ailleurs conseillé pour clarifier des calculs qui seraient peu lisibles autrement.

2.7.6 L'abstraction : Ne jamais écrire deux fois la même chose

Un des concepts les plus importants en programmation est *l'abstraction*, c'est-à-dire la capacité à trouver des formes générales à partir de plusieurs formes similaires⁶.

Méthode

La méthode de l'abstraction consiste à d'abord écrire "à la main" plusieurs exemples du résultat que l'on souhaite obtenir. Une fois fait, ceci permet de :

- i - Identifier les différences entre les exemples,
- ii - Identifier les similarités entre les exemples,
- iii - Utiliser une variable pour chacune des différences et réécrire le code en remplaçant les occurrences des différences par l'utilisation des variables. Pour aider à la création du code, donner aux variables les valeurs qu'elles doivent prendre sur le premier exemple,
- iv - Transformer le code précédent en une fonction prenant en argument toutes les variables ainsi trouvées, Les variables ne doivent donc plus avoir les valeurs du premier exemple. La *valeur de retour* de la fonction est soit la valeur calculée, soit aucune s'il s'agit d'une fonction d'affichage.

6. En intelligence artificielle, on parle aussi d'*induction*, qui est considérée comme l'un des mécanismes centraux de l'intelligence.

- v - Si la fonction précédente est une fonction d'affichage, il est préférable de la transformer pour lui faire retourner une chaîne de caractères,
- vi - Finalement faire des appels à cette fonction sur les différents exemples, en lui passant en argument les différences des exemples.

Déroulement sur un exemple

Nous possédons les deux exemples suivants que nous voulons afficher en sortie :

```
Nord, Nord-Nord-Ouest, Nord-Ouest, Ouest-Nord-Ouest, Ouest.
```

```
Sud, Sud-Sud-Est, Sud-Est, Est-Sud-Est, Est.
```

Commençons par écrire le code qui permet d'afficher ces deux chaînes :

```
1 print('Nord, Nord-Nord-Ouest, Nord-Ouest, Ouest-Nord-Ouest, Ouest.');
```

et :

```
1 print('Sud, Sud-Sud-Est, Sud-Est, Est-Sud-Est, Est.');
```

C'est ce code que nous allons modifier pour obtenir une abstraction.

i - Identification des différences entre les exemples

Le mot 'Nord' est remplacé par le mot 'Sud' et le mot 'Ouest' est remplacé par le mot 'Est'.

Ces mots seront donc remplacés par des variables `$dir1` et `$dir2`.

ii - Identification des similarités entre les exemples

Ici, il s'agit de l'enchaînement des mots et de la ponctuation.

iii - Utilisation de variables

Réécrivons le code ci-dessus en remplaçant le premier exemple par les variables identifiées :

```
1 $dir1 = 'Nord';
2 $dir2 = 'Ouest';
3 print($dir1.', ' .
4     $dir1.'-'.'$dir1.'-'.'$dir2.', ' .
5     $dir1.'-'.'$dir2.', ' .
6     $dir2.'-'.'$dir1.'-'.'$dir2.', ' .
7     $dir2.'.'');
```

iv - Création de la fonction

Les deux variables passent en argument de notre nouvelle fonction, à la quelle nous don-

nous le nom de `afficher_cardinaux` :

```

1 function afficher_cardinaux($dir1, $dir2) { // string string -> none
2     print($dir1.', '.
3         $dir1.'-'.'$dir1.'-'.'$dir2.', '.
4         $dir1.'-'.'$dir2.', '.
5         $dir2.'-'.'$dir1.'-'.'$dir2.', '.
6         $dir2.'.'');
7 }

```

Notez le commentaire à droite du nom de la fonction signalant que les deux arguments sont de type "string", et que la fonction ne renvoie aucune valeur.

v - Valeur de retour

On supprime le `print` de la fonction :

```

1 function points_cardinaux($dir1, $dir2) { // string string -> string
2     return $dir1.', '.
3         $dir1.'-'.'$dir1.'-'.'$dir2.', '.
4         $dir1.'-'.'$dir2.', '.
5         $dir2.'-'.'$dir1.'-'.'$dir2.', '.
6         $dir2.'.' ;
7 }

```

Notez :

- l'utilisation du `return` à la place du `print`,
- Le changement de nom de la fonction,
- Le changement du commentaire à droite du nom de fonction.

vi - Utilisation de la fonction

Il ne nous reste plus qu'à effectuer des appels à notre nouvelle fonction :

```

1 <?php
2 function points_cardinaux($dir1, $dir2) { // string string -> string
3     return $dir1.', '.
4         $dir1.'-'.'$dir1.'-'.'$dir2.', '.
5         $dir1.'-'.'$dir2.', '.
6         $dir2.'-'.'$dir1.'-'.'$dir2.', '.
7         $dir2.'.' ;
8 }
9
10 function main() {
11     printline( points_cardinaux('Nord', 'Ouest') );
12     printline( points_cardinaux('Sud', 'Est') );
13 }
14 ?>

```

Avantages de l'abstraction

Maintenant que la fonction est créée, il est beaucoup plus facile d'effectuer des modifications sur les *ressemblances* des exemples. Par exemple, si l'on veut changer les "," par des ";", il suffit de les changer dans la fonction, indépendamment du nombre de fois où cette fonction est appelée. Sans cette abstraction, on aurait été obligé de le modifier dans chaque exemple, avec un risque croissant d'erreur.

L'*effort de modification* et le risque d'erreur sont ainsi diminués, et la facilité de réutilisation ultérieure est grandement augmenté : au lieu de faire un copier/coller risquant une fois de plus d'engendrer des erreurs (généralement des oublis de modification), il suffit de faire un appel unique de fonction. La taille du programme s'en trouve par la suite aussi grandement diminuée, ce qui contribue à le rendre plus clair, de même que de choisir un nom de fonction adéquat.

Généralisation de l'abstraction

Même lorsqu'il semble trop "lourd" de créer une fonction, il est néanmoins possible d'effectuer une abstraction, en utilisant par exemple des variables, ou des boucles comme nous le verrons plus tard, contribuant ainsi à ne jamais écrire deux fois la même chose.

Prenons le code suivant :

```
1 $chaine = 'philanthrope philanthrope';
```

bien que l'on ne possède pas ici deux exemples pour déterminer les ressemblances et les différences entre eux, on peut néanmoins noter la réutilisation d'un mot. On peut ainsi s'abstraire de ce mot et écrire plutôt :

```
1 $mot = 'philanthrope';
2 $chaine = $mot . ' ' . $mot;
```

2.7.7 Méthode diviser pour résoudre

Nous détaillons ici une méthode générale pour modéliser puis programmer la solution à un problème dont l'énoncé est donné.

Cette méthode est utile non seulement en informatique, mais aussi dans toute discipline où il est besoin de résoudre un problème d'une certaine complexité.

Supposez que vous avez un problème opératoire à résoudre, c'est-à-dire un problème nécessitant une séquence d'actions pour arriver à la solution (ce qui, notez bien, est le cas pour à peu près tous les problèmes).

La programmation, grâce à sa capacité à automatiser des opérations, sert à résoudre des problèmes généraux, pour éviter de répéter sa résolution "à la main" à chaque fois qu'une variante de ce problème survient.

L'idée principale est de décomposer le problème en sous-problèmes, en utilisant les fonctions.

Nous allons expliquer la méthode de résolution sur le problème suivant :

On souhaite pouvoir calculer le salaire net réel annuel d'un employé à temps plein de la société Cacoloc, étant donné son salaire horaire net et un indice des prix⁷.

Donner un nom à la fonction

Commencez par donner un nom *explicite* à la fonction qui doit résoudre le problème.

Ajoutez un commentaire pour dire ce qu'est censée faire la fonction (description succincte du problème).

Pour notre exemple :

```
1 // Calcule le salaire annuel réel à partir du salaire horaire net
2 // et d'un indice des prix.
3 fonction salaire_annuel_reel () {
4 }
```

Identifier le type de retour de la fonction

Une fois cette fonction appelée, celle-ci doit retourner une valeur. Quelle est le *type* de cette valeur ? (chaîne de caractères, entier, réel, booléen, etc.)

Dans notre exemple, la fonction `salaire_annuel_reel` retourne un nombre réel :

```
1 // Calcule le salaire annuel réel à partir du salaire horaire net
2 // et d'un indice des prix.
3 fonction salaire_annuel_reel () { // -> real
4 }
```

Le signe `-> real` signifie que le type de retour de la fonction est un réel.

Identifier les arguments d'entrée de la fonction

Pour cela, il faut commencer par faire une liste des valeurs utilisées dans la fonction. Pour chacune des entrées, déterminer si c'est une valeur qui changera ou non selon les différentes versions du problème (les différents appels à la fonction). Si cette valeur change, alors c'est un argument de la fonction.

Pour aider à déterminer ces arguments, il est utile d'écrire 2 ou 3 cas d'utilisation de la fonction, en mettant en commentaire la valeur que doit retourner la fonction. Quelles sont les valeurs qui changent lors de ces différents appels ?

Il faut aussi identifier les types des arguments.

Arrivé ici, on obtient une *signature* de fonction.

7. $\text{salaire réel annuel} = \text{salaire net annuel} / \text{indice des prix}$

Dans notre exemple :

```
1 // Calcule le salaire annuel réel à partir du salaire horaire net
2 // et d'un indice des prix.
3 fonction salaire_annuel_reel($sal_h, $ind_prix) { // real real -> real
4 }
```

Ici `$sal_h` et `$ind_prix` sont tous les deux des réels, d'où le nouveau commentaire de la fonction : `real real -> real`.

Quelques cas d'utilisation :

```
1 printline(salaire_annuel_reel(15.0, 1.2)); // -> 22750.0
2 printline(salaire_annuel_reel(49.5, 1.05)); // -> 85800.0
```

Les `printline` permettent de comparer la valeur affichée avec la valeur calculée à la main.

Identifier les sous-problèmes

Décomposez votre problème en 1 ou 2 sous-problèmes indépendants (ou plus si nécessaire). Chaque sous-problème est lui-même résolu par une fonction, donc pour chaque sous-problème, répétez le processus ci-dessus :

- Donner un nom à la (sous-)fonction, accompagné d'une description générale,
- Identifier le type de retour de la fonction,
- Identifier les arguments d'entrée, et écrire 2 ou plus cas d'utilisation, et écrire la signature de la fonction.

Notez bien que pour le moment on ne s'intéresse pas à la manière dont ces sous-problèmes sont résolus !

Dans notre exemple, nous identifions deux sous-problèmes :

- Calculer le salaire net annuel à partir du salaire net horaire,
- Calculer le salaire réel à partir du salaire annuel.

```
1 <?php
2 // Calcule le salaire net annuel à partir du salaire net horaire.
3 function salaire_annuel_net($sal_h) { // real -> real
4 }
5
6 // Calcule le salaire réel à partir du salaire net
7 // et d'un indice des prix.
8 function net_vers_reel($sal_an, $ind_prix) { // real real -> real
9 }
10
11 // Calcule le salaire annuel réel à partir du salaire horaire net
12 // et d'un indice des prix.
13 function salaire_annuel_reel($sal_h, $ind_prix) { // real real -> real
14 }
15
16 function main() {
17     // Nouveaux cas d'utilisation :
18     printline(salaire_annuel_net(10.0)); // -> 18200.0
19     printline(salaire_annuel_net(49.5)); // -> 90090.0
20
21     printline(net_vers_reel(22000, 1.6)); // -> 13750.0
22     printline(net_vers_reel(35000, 0.8)); // -> 43750.0
23
24     printline(salaire_annuel_reel(15.0, 1.2)); // -> 22750.0
25     printline(salaire_annuel_reel(49.5, 1.05)); // -> 85800.0
26
27 }
28 ?>
```

Remplir le corps de la fonction

Bien que l'on n'ait pas encore écrit le code des sous-fonctions, on considère que l'on sait qu'elles fonctionnent correctement, sans avoir besoin de savoir comment.

Sachant cela, on remplit le corps de la fonction avec des appels aux différentes sous-fonctions.

Ceci est la partie la plus délicate, mais elle devrait être largement simplifiée si vous avez suivi les étapes ci-dessus.

```

1 <?php
2 ...
3
4 // Calcule le salaire annuel réel à partir du salaire horaire net
5 // et d'un indice des prix.
6 function salaire_annuel_reel($sal_h, $ind_prix) { // real real -> real
7     $sal_an = salaire_annuel_net($sal_h);
8     $sal_reel = net_vers_reel($sal_an, $ind_prix);
9     return $sal_reel;
10 }
11
12 ...
13 ?>

```

Décomposer les sous-problèmes

Enfin, les sous-problèmes eux-mêmes peuvent être décomposés en sous-problèmes si besoin (et ainsi de suite), de manière à pouvoir remplir le code de leur fonction associée.

Dans notre exemple, les sous-problèmes sont suffisamment simples pour être résolus immédiatement. On obtient alors :

```

1 <?php
2 // Calcule le salaire net annuel à partir du salaire net horaire.
3 function salaire_annuel_net($sal_h) { // real -> real
4     return $sal_h * 35 * 52; // 35h/semaine, 52 semaines
5 }
6
7 // Calcule le salaire réel à partir du salaire net
8 // et d'un indice des prix.
9 function net_vers_reel($sal_an, $ind_prix) { // real real -> real
10    return $sal_an / $ind_prix;
11 }
12
13 // Calcule le salaire annuel réel à partir du salaire horaire net
14 // et d'un indice des prix.
15 function salaire_annuel_reel($sal_h, $ind_prix) { // real real -> real
16    $sal_an = salaire_annuel_net($sal_h);
17    $sal_reel = net_vers_reel($sal_an, $ind_prix);
18    return $sal_reel;
19 }
20
21 function main() {
22     // Nouveaux cas d'utilisation :
23     printline(salaire_annuel_net(10.0)); // -> 18200.0
24     printline(salaire_annuel_net(49.5)); // -> 90090.0
25
26     printline(net_vers_reel(22000, 1.6)); // -> 13750.0
27     printline(net_vers_reel(35000, 0.8)); // -> 43750.0

```

```
28
29     printline(salaire_annuel_reel(15.0, 1.2)); // -> 22750.0
30     printline(salaire_annuel_reel(49.5, 1.05)); // -> 85800.0
31
32 }
33 ?>
```

Dès qu'un sous-problème est entièrement résolu, il faut le *tester*, c'est-à-dire vérifier qu'il fonctionne correctement en exécutant le fichier PHP et en vérifiant les valeurs obtenues.

2.7.8 Fonctions, Abstraction et Diviser pour résoudre : Conclusion

En combinant le raisonnement par abstraction (partir de quelques exemples d'utilisation pour créer une méthode générale), et de la méthode *diviser pour résoudre*, il est possible de venir à bout de tout problème de programmation, quelque soit sa complexité : l'idée majeure étant de décomposer ce problème en sous-problèmes, de résoudre les (sous-)problèmes un par un en utilisant une fonction pour chaque.

Si vous acquérez bien cette méthode, vous aurez un sérieux atout pour résoudre aussi n'importe quel problème d'ingénierie.

2.8 La boucle while

"Comment afficher les nombres de 1 à 1000, sans les écrire tous un par un ?"

Le principe des *boucles* est de répéter un même schéma un certain nombre de fois d'affilé jusqu'à une *condition d'arrêt*. Comme dans une recette de cuisine, "battre les oeufs en neige" signifie "faire plusieurs fois (battre les oeufs) jusqu'à (les oeufs sont montés en neige)", où ici le schéma répétitif est le fait de battre une seule fois les oeufs.

Pour créer des boucles, il faut donc **repérer le schéma répétitif** et **identifier la condition d'arrêt**.

En PHP, toute tâche répétitive peut se réaliser avec la *boucle while*. Le schéma général de la boucle **while** est le suivant :

```
1 while(condition) {
2     instruction
3     instruction
4     ...
5 }
```

On remarque immédiatement la similarité avec la condition **if**, mais contrairement à ce dernier qui n'est pas une boucle, le **while** effectue plusieurs fois un groupe d'instructions, *tant que* la condition *condition* est vérifiée.

En français, **while** signifie **tant que**, et il faut réellement le penser comme cela :

```
1 tant que (la condition est vérifiée) faire
2     des instructions
```

Pour battre les oeufs en neige, il faut écrire :

```
1 tant que (les oeufs ne sont pas montés en neige) faire
2     battre les oeufs
```

où la condition est mise cette fois sous forme *négative* : le "tant que" est la forme négative du "jusqu'à ce que". Réfléchissez donc bien à votre condition de boucle ; cela aide généralement de l'exprimer clairement en français.

Voici en exemple la boucle permettant d'afficher les nombres de 1 à 20 :

```
1 $i = 1;
2
3 while($i <= 20) {
4     print($i . ' ');
5     $i++;
6 }
7
8 printline('Fin !');
```

Ce qui produit le résultat suivant :

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 Fin !
```

Juste avant la boucle, on commence par donner une *valeur initiale* à la variable `$i`. On arrive ensuite sur la boucle `while`.

Avant même d'entrer pour la première fois dans la boucle pour exécuter les instructions répétitives, **on commence par tester la condition**. C'est pour cela qu'elle se trouve au dessus des instructions répétitives. Si la condition n'est pas vérifiée (sa valeur est `false`), on *sort* immédiatement de la boucle, sans même avoir effectué une seule fois les instructions de boucle.

Sortir de la boucle signifie passer aux instructions qui suivent l'instruction ou le bloc d'instructions de la boucle, en l'occurrence `printline('Fin !');`.

Si par contre la condition est vérifiée (la valeur calculée est `true`), on entre dans la boucle pour exécuter les instructions répétitives, toujours selon leur ordre : on commence par afficher `$i` qui a la valeur 1, puis on *incrémente* `$i`, c'est-à-dire qu'on lui ajoute 1. Arrivé là, les instructions répétitives de la boucle sont finies pour le moment.

Lorsque l'on a exécuté une fois les instructions de boucle, on *remonte* sur la condition et on la teste à nouveau. Idem, si elle n'est pas vérifiée, on sort de la boucle et on arrête d'exécuter les instructions de boucle. Sinon, si la condition est vérifiée, on exécute à *nouveau* les instructions de boucle, toujours dans le même ordre. Puis on remonte à nouveau sur la condition, que l'on teste à nouveau, et ainsi de suite jusqu'à ce que la condition ne soit plus vérifiée.

Le programme exécute donc les lignes d'instructions dans cet ordre précis : 1, 3, 4, 5, 3, 4, 5, ... 3, 4, 5, 8. Notez bien que la dernière instruction de la boucle `while` qui est exécutée est la condition, puisque c'est elle qui dit quand s'arrêter.

Si jamais on met une mauvaise condition à la boucle, on risque soit de ne jamais entrer dans la boucle :

```
1 $i = 1;
2
3 while($i < 0) {
4     print($i . ' ');
5     $i++;
6 }
7
8 printline('la valeur de i est ' . $i);
```

ce qui affichera :

```
la valeur de i est 1
```

soit de ne jamais sortir de la boucle :

```
1 $i = 1;
2
3 while($i > 0) {
4     print($i . ' ');
5     $i++;
6 }
7
8 printline('Ceci ne sera jamais affiché.');
```

ce qui écrira tous les nombres sans jamais arriver à la dernière ligne du programme.

2.8.1 Imbrications

Dans un bloc d'instructions on peut mettre tout ce que l'on a vu jusqu'à maintenant : des affectations, des conditions, des affichages, des boucles, ... Donc, tout comme pour le `if`, on peut imbriquer des boucles et des conditions `if`, ou des boucles et des boucles, etc.

Exercice 2 *Que fait alors le programme suivant ?*

```
1 <?php
2 function main() {
3     $x = 1;
4
5     while($x <= 1000) {
6         $n = 1;
7
8         while($n <= 5) {
9             print($x . ' ');
10
11             $n++;
12         }
13
14         $x++;
15     }
16 }
17 ?>
```

Remarquez bien que l'instruction `$n = 1;` est placée à l'intérieur de la première boucle `while`, et sera donc exécutée pour *chaque* nouvelle valeur de `$x`.

Si vous avez des difficultés à voir ce que fait ce programme, prenez un papier et un crayon et exécutez le programme à la main (au moins pour les premières valeurs) et notez à chaque instruction les valeurs de `$x` et `$n`.

2.9 La boucle do...while

La boucle `do...while` fait exactement la même chose que la boucle `while`, à la seule différence que la condition est testée *après* les instructions répétitives :

```
1 do {
2     instruction
3     instruction
4     ...
5 } while(condition);
```

*Attention, cette fois il y a bien un ; à la fin du **while**!*

La boucle **do...while** garantit que l'on exécutera au moins une fois les instructions de boucle, ce qui peut être pratique dans certains cas pour éviter de réécrire les instructions au dessus de la boucle.

Elle n'est cependant pas couramment utilisée.

2.10 La boucle for

La boucle **for** n'est pas nécessaire en soi, puisque l'on pourrait toujours utiliser la boucle **while**, mais c'est une manière plus pratique et très courante d'écrire certains types de boucles. Elle est généralement utilisée lorsque l'on connaît à l'avance le nombre de fois où la boucle doit être exécutée.

Sa forme générale est la suivante :

```
1 for(initialisation ; condition ; incrémentation) {
2     instruction
3     instruction
4     ...
5 }
```

et elle est strictement équivalente à la forme suivante du **while** :

```
1 initialisation
2
3 while(condition) {
4     instruction
5     instruction
6     ...
7     incrémentation
8 }
```

L'initialisation et la condition sont donc exécutées avant toute exécution des instructions répétitives. L'incrémentation, elle, est exécutée après, juste avant de tester à nouveau la condition.

Par exemple, le code précédent :

```
1 $i = 1;
2
3 while($i <= 1000) {
4     print($i . ' ');
5     $i++;
6 }
7
8 printline('Fin !');
```

peut aussi s'écrire :

```
1 for($i = 1; $i <= 1000; $i++) {
2     print($i . ' ');
3 }
4
5 printline('Fin !');
```

Comme pour le **if** et le **while**, il n'y a pas de ; à la fin de la ligne du **for**. Par contre, notez bien les deux ; à l'intérieur des parenthèses du **for**, pour séparer les différents types d'instructions.

Exercice 3 *Transformer le programme de l'exercice 2 contenant les deux boucles **while** en un programme n'utilisant que des boucles **for**.*

2.11 Tableaux

Les tableaux sont le dernier élément essentiel des langages de programmation. Ce sont des genres de variables spéciales qui permettent de stocker une *suite de variables*.

On peut donc les voir comme une suite de cases mémoires, chacune contenant une valeur. Chaque case est associée à un *indice* qui permet d'accéder à la case correspondante. En PHP comme dans de nombreux langages, le premier indice est 0.

Indice :	0	1	2	3	4	5	6	7	8	9
Valeur :	42	51	84	93	1	77	93	80	56	81

Un tableau, tout comme une variable, peut bien sûr contenir autre chose que des valeurs numériques :

Indice :	0	1	2
Valeur :	'rouge'	'vert'	'bleu'

En PHP, pour déclarer un tableau, on utilise le mot clef **array**.

```
1 // Déclaration et premier remplissage du tableau :
2 $nombres = array(42, 51, 84, 93, 1, 77, 93, 80, 56, 81);
```

Les valeurs sont rangées dans l'ordre où elles sont données, et les indices sont mis automatiquement. Le tableau **\$nombres** est aussi une variable, bien qu'il contienne plusieurs valeurs, c'est pourquoi il est précédé du signe **\$**.

Si plus tard on veut ajouter des valeurs à la fin de ce tableau, on peut utiliser la notation **"[]"** collée au nom du tableau à gauche de l'affectation :

```
1 // Déclaration et premier remplissage du tableau :
2 $couleurs = array('rouge', 'vert', 'bleu');
3
4 // Ajout de la valeur 'jaune' en fin de tableau :
5 $couleurs[] = 'jaune';
6
7 // Ajout de la valeur 'cyan' en fin de tableau
8 $couleurs[] = 'cyan';
```

À la fin de ce programme, le tableau **\$nombres** contiendra donc les valeurs :

Indice :	0	1	2	3	4
Valeur :	'rouge'	'vert'	'bleu'	'jaune'	'cyan'

Pour accéder à une case particulière d'un tableau, on utilise la même notation **"[]"** que précédemment, mais cette fois en spécifiant l'indice de la case souhaitée :

```
1 $couleurs2 = array('rouge', 'vert', 'bleu', 'jaune', 'cyan');
2
3 printline($couleurs2[2] . ', ' . $couleurs2[4]);
```

Ce code affichera donc :

```
bleu, cyan
```

De la même manière, on peut modifier le contenu des cases :

```
1 $couleurs = array('rouge', 'vert', 'bleu');
2
3 $couleurs[0] = 'blanc';
4
5 $couleurs[2] = 'noir';
```

À la fin de ce programme, le contenu du tableau `$couleurs` est :

Indice :	0	1	2
Valeur :	'blanc'	'vert'	'noir'

2.11.1 Parcours en boucle

Pour parcourir chacune des cases d'un tableau (par exemple pour les afficher ou les modifier), on utilise une boucle.

```
1 $couleurs2 = array('rouge', 'vert', 'bleu', 'jaune', 'cyan');
2
3 // Affichage de toutes les valeurs du tableau :
4 for($indice = 0; $indice < count($couleurs2); $indice++) {
5     $valeur = $couleurs2[$indice];
6     printline($valeur . ', ');
7 }
```

où `count` est une fonction qui retourne le nombre d'éléments que contient le tableau qui lui est passé en paramètre. La variable `$indice` va prendre successivement les valeurs 0, 1, 2... jusqu'à la taille du tableau - 1 (à cause du signe "<" qui fait arrêter la boucle dès que l'on a `$indice == count($couleurs2)`).

Exercice 4 *Écrire une boucle qui ajoute 1 à chaque valeur du tableau `$nombres`.*

2.11.2 La boucle foreach

La boucle `foreach` offre une manière plus pratique pour parcourir un tableau qu'une boucle `for` ou une boucle `while`. Voici un exemple d'utilisation, équivalent au dernier programme :

```
1 $couleurs2 = array('rouge', 'vert', 'bleu', 'jaune', 'cyan');
2
3 foreach($couleurs2 as $valeur) {
4     printline($valeur . ', ');
5 }
```

Il signifie qu'on parcourt le tableau `$couleurs` avec la (nouvelle) variable `$valeur` qui prend successivement toutes les valeurs du tableau à chaque itération de la boucle. Il n'y a pas besoin de faire explicitement l'affectation suivante :

```
1 $valeur = $couleurs2[$indice];
```

car elle est faite implicitement par le **foreach**.

Attention, supposons que l'on veuille mettre toutes les valeurs du tableau à 'blanc'; on ne peut pas faire, à l'intérieur de la boucle :

```
1 $valeur = 'blanc';
```

car on ne ferait que modifier la valeur de la variable **\$valeur** sans toucher directement au contenu du tableau.

Dans ce cas, on peut par contre utiliser une variable qui parcourt les indices du tableau (0, 1, 2...) avec la construction suivante du **foreach** :

```
1 $couleurs2 = array('rouge', 'vert', 'bleu', 'jaune', 'cyan');
2
3 foreach($couleurs2 as $indice => $valeur) {
4     printline('indice : ' $indice);
5     // Affichage de l'ancienne valeur dans le tableau :
6     printline('valeur : ' $valeur);
7     // Modification de cette valeur :
8     $couleurs2[$indice] = 'blanc';
9 }
```

2.11.3 Tableaux associatifs

Les précédents tableaux sont nommés tableaux scalaires, parce qu'ils utilisent des nombres pour les indices. Il existe une autre forme de tableaux, nommés *tableaux associatifs*, où les cases mémoires ne sont pas rangées par ordre croissant d'indice. On ne repère donc pas les indices des cases par leur numéro, mais par une *clef* qui est une chaîne de caractères. Voici un exemple :

```
1 $capitales = array();
2 // Ajout d'une nouvelle valeur et de sa clef associée :
3 $capitales['France'] = 'Paris';
4 // Idem :
5 $capitales['R.U.'] = 'Londres';
6 // Idem :
7 $capitales['Allemagne'] = 'Berlin';
```

Ceci produira le tableau **\$capitales** suivant :

Indice :	'France'	'R.U.'	'Allemagne'
Valeur :	'Paris'	'Londres'	'Berlin'

Les clefs sont ici les pays, au même titre que les nombres étaient les indices pour les tableaux scalaires. Les valeurs sont les capitales.

On peut écrire aussi directement les associations lors de la déclaration de la variable

\$capitales comme tableau :

```
1 $capitales = array( 'France' => 'Paris',  
2                   'R.U.' => 'Londres',  
3                   'Allemagne' => 'Berlin' );
```

Remarquez l'utilisation de la flèche =>, qui est exactement la même que pour le parcours dans le **foreach** avec indice et valeur.

D'ailleurs, le parcours d'un tel tableau se fait logiquement avec la boucle **foreach** :

```
1 $capitales = array( 'France' => 'Paris',  
2                   'R.U.' => 'Londres',  
3                   'Allemagne' => 'Berlin' );  
4  
5 foreach($capitales as $clef => $valeur) {  
6     printline('Ville : ' . $clef . ' ; Capitale : ' . $valeur);  
7 }
```

où cette fois les indices sont les clefs. Pour les tableaux associatifs, il est plus courant d'utiliser la forme complète du **foreach** car on a plus souvent à la fois des valeurs et des clefs associées.

Chapitre 3

Le paquetage AgroSIXPack

3.1 Introduction

Le langage PHP a été initialement conçu pour la création de sites Web, interagissant pour cela avec le langage HTML (*HyperText Markup Language*) de description de pages Web.

Dans ce cours, nous ne souhaitons pas entrer dans les détails du langage HTML, qui n'est que d'un intérêt pédagogique relatif.

C'est pourquoi nous avons développé le paquetage AgroSIXPack (pour *AgroParis-Tech Systèmes d'Information eXtension Package*), qui permet de masquer entièrement le langage HTML et de se concentrer sur la partie PHP.

Sachez néanmoins que HTML existe et que le langage PHP que vous allez voir maintenant est une variante du PHP habituel (de base, pourrait-on dire).

Il existe aussi des Systèmes de Gestion de Contenu (*Content Management System*, CMS) permettant de créer des sites Web sans réelle notion de programmation¹. Ces CMS sont très souvent basés sur un langage de programmation tel que Java ou PHP.

Le paquetage AgroSIXPack est un genre de petit CMS, mais permettant de se focaliser sur la partie programmation simple sans entrer dans les détails parfois complexes de la création de site Web.

3.2 Fonction principale

Pour utiliser ce paquetage, il faut télécharger l'archive AgroSIXPack.zip (voir le site web du cours), la décompresser dans le répertoire N : \www.

Tout fichier .php exécuté par AgroSIXPack doit définir la fonction `main` :

1. Enfin, tant que l'on se cantonne aux fonctionnalités prévues, car la connaissance de la programmation (parfois avancée !) est très vite nécessaire dès que l'on veut faire des sites un peu particuliers ou plus évolués.

```

1 function : main( $id_formulaire )
2   |   $id_formulaire : string

```

Elle doit prendre en argument la variable `$id_formulaire`, qui représente le `$nom` du formulaire qui a envoyé les données (entraînant l'exécution de ce fichier PHP), ou bien la chaîne vide " " si l'exécution de ce fichier n'est pas due à l'envoi de données par un formulaire, par exemple en cliquant sur un lien ou en tapant l'URL de la page directement dans le navigateur. Cet argument peut être omis s'il n'est pas utilisé.

Cette fonction (et seulement celle-ci) est exécutée automatiquement par le paquetage au lancement de la page Web dans le navigateur.

Il faut ensuite ouvrir le fichier `AgroSIXPack.php` à travers le serveur PHP, soit (probablement, selon votre configuration) à l'adresse `http://localhost/www/PHP/AgroSIXPack.php`.

Remarque : Ce chapitre contient de nombreuses signatures de fonction, reportez-vous à l'annexe page 86 pour savoir comment les lire.

3.3 Affichage

Attention : Dans tout ce qui suit, il est extrêmement important de bien faire la distinction entre *affichage d'une valeur à l'écran* et *renvoi d'une valeur par retour d'un appel de fonction*. Seul le second cas permet de *recupérer* cette valeur pour la *réutiliser* dans un calcul ultérieur. L'affichage à l'écran ne sert réellement qu'à afficher des informations à l'utilisateur, pas plus, et n'a aucune utilité par rapport au calcul que l'on cherche à faire.

```

1 function : print( $val )
2   |   $val : any

```

Cette fonction est une des fonctions de base de PHP, elle permet simplement d'afficher une valeur à l'écran, que ce soit une chaîne de caractères ou un nombre, mais pas des tableaux.

```

1 function : printline([ $texte ] ) -> string
2   |   $texte : string = ''

```

Comme `print` mais passe à la ligne ensuite.

```

1 function : newline() -> string

```

Renvoient une chaîne de caractères correspondant à un passage à la ligne. Cette fonction doit donc être utilisée dans une fonction d'affichage telle que `print` pour que le passage à la ligne soit visible à l'écran.

```

1 function : afficher_titre( $texte [, $niveau ] )
2   | $texte : string
3   | $niveau : int      = 1

```

Affiche une chaîne de caractères `$texte` avec un titre de niveau `$niveau` (de 1 à 6).

```

1 function : afficher_hrule()

```

Affiche un ligne horizontale.

```

1 function : bold( $texte ) -> string
2   | $texte : string

```

```

1 function : emph( $texte ) -> string
2   | $texte : string

```

Renvoie la chaîne de caractères `$texte` en gras ou en italique.

```

1 function : debut_liste( [ $nom ] )
2   | $nom : string = ''

```

```

1 function : afficher_liste_item( $texte )
2   | $texte : string

```

```

1 function : fin_liste( [ $nom ] )
2   | $nom : string = ''

```

Affichent une liste d'items. Les listes peuvent être imbriquées pour créer plusieurs niveaux, mais il doit toujours y avoir le même nombre d'appels à `debut_liste`.

```

1 function : debut_table( [ $nom ] )
2   | $nom : string = ''

```

```

1 function : fin_table( [ $nom ] )
2   | $nom : string = ''

```

Un appel à ces deux fonctions doit encadrer tout affichage de table. Les tables ne peuvent pas être imbriquées.

```

1 function : afficher_table_ligne( $ligne )
2   | $ligne : array(string)

```

Affiche une ligne entière de données, à partir du tableau `$ligne`. Un appel à cette fonction

ne doit **pas** être encadré d'appels à `debut_table_ligne` et `fin_table_ligne`.

```
1 function : afficher_table_entete( $ligne )
2 | $ligne : array(string)
```

Comme `afficher_table_ligne` mais pour les entêtes des colonnes de la table. Cette fonction ne peut-être appelée qu'une seule fois par table, et avant toute autre ligne.

```
1 function : debut_table_ligne()
```

```
1 function : afficher_table_donnee( $data )
2 | $data : string
```

```
1 function : fin_table_ligne()
```

Comme `afficher_table_ligne` mais décomposé en 3 fonctions. La fonction `afficher_table_donnee` doit être appelée autant de fois qu'il y a de cellules dans un ligne. Un ensemble d'appels doit être encadré des appels à `debut_table_ligne` et `fin_table_ligne`.

```
1 function : afficher_image( $url [, $texte_alt ] )
2 | $url      : string
3 | $texte_alt : string = ''
```

Affiche une image. L'`$url` doit être soit une adresse internet commençant par "http ://", soit un fichier local dont l'adresse est *relative* au dossier courant. Le texte `$texte_alt` est le texte affiché lorsque la souris est posée sur l'image.

```
1 function : afficher_lien_interne( $texte , $page )
2 | $texte : string
3 | $page  : string
```

Affiche un lien interne vers une `$page` PHP relativement au répertoire courant.

```
1 function : afficher_lien_externe( $texte , $url )
2 | $texte : string
3 | $url   : string
```

Affiche un lien vers une URL externe (commençant par "http ://" ou "https ://").

3.4 Formulaires : Interaction avec l'utilisateur

Jusque là, tout ce que nous avons vu en PHP ne peut nous servir qu'à faire du calcul ou à les afficher à l'écran. Nous allons maintenant voir comment interagir avec l'utilisateur, en lui demandant des valeurs, qui vont donc permettre de faire des calculs différents selon

les valeurs récupérées.

Pour cela, nous allons utiliser des *formulaires*. Il s'agit d'un ensemble d'outils permettant à l'utilisateur de saisir des valeurs ou de faire des choix, que l'on pourra prendre en compte dans les calculs subséquents.

Voici un exemple de formulaire, montrant l'utilisation de la plupart des fonctions que nous allons décrire par la suite :

- premier
- second
- troisième

a	b	c
1	2	3
x	y	z

Prenom :

Nom :

Texte :

Choix : ▾

Quels animaux possédez-vous ?

Chat

Chien

Lapin

Dors beaucoup ?

Quelle radio écoutez-vous?

RFM

France Info

Oui FM

Et voici le code PHP correspondant :

```

1 <?php
2
3 function afficher_liste_et_table() {
4     debut_liste();
5     afficher_liste_item('premier');
6     afficher_liste_item('second');
7     afficher_liste_item('troisième');
8     fin_liste();
9
10    debut_table();
11    afficher_table_ligne( array('a', 'b', 'c') );
12
13    afficher_table_ligne( array('1', '2', '3') );
14
15    debut_table_ligne();

```

```
16         afficher_table_donnee('x');
17         afficher_table_donnee('y');
18         afficher_table_donnee('z');
19     fin_table_ligne();
20
21     fin_table();
22
23 }
24
25 function afficher_formulaire() {
26     debut_formulaire('form1', 'traitement.php');
27
28     afficher_champ_texte('prenom', 'Prenom : ', 'John');
29     afficher_champ_texte('nom', 'Nom : ', 'Arbuckle');
30
31     afficher_champ_paragraphe('para', 'Texte : ', "Bonjour\nnet
    bienvenue", 4, 23);
32
33     debut_select('liste', 'Choix : ');
34     afficher_select_item('premier'); // valeur envoyée si
    choisi : premier
35     afficher_select_item('second', 2); // valeur envoyée si
    choisi : 2
36     fin_select();
37
38     debut_groupe_checkbox('animaux', 'Quels animaux possédez-vous ?
    ');
39     afficher_checkbox_item('chat', 'Chat');
40     afficher_checkbox_item('chien', 'Chien', true);
41     afficher_checkbox_item('lapin', 'Lapin', true);
42     fin_groupe_checkbox('animaux');
43
44     afficher_checkbox('dors', 'Dors beaucoup ?', true);
45
46     debut_groupe_radio('maradio', 'Quelle radio écoutez-vous?');
47     afficher_radio_item('rfm', 'RFM', false);
48     afficher_radio_item('frinfo', 'France Info', false);
49     afficher_radio_item('oui', 'Oui FM', true);
50     fin_groupe_radio();
51
52
53     fin_formulaire('form1', 'Envoyer');
54 }
55
56 function main() {
57     afficher_liste_et_table();
58     printline();
59     afficher_formulaire();
60 }
61
62 ?>
```

3.4.1 Création et affichage de formulaires

Un formulaire doit obligatoirement commencer par un appel à la fonction :

```
1 function : debut_formulaire( $id_form , $page )
2 |   $id_form : string
3 |   $page    : string
```

où `$id_form` est un nom interne du formulaire, et `$page` est le nom de la page PHP qui recevra les données du formulaire lorsque l'utilisateur aura cliqué sur le bouton 'OK'.

Un formulaire doit également obligatoirement finir par un appel à la fonction :

```
1 function : fin_formulaire( $id_form [, $bouton_ok_str ] )
2 |   $id_form      : string
3 |   $bouton_ok_str : string = 'OK'
```

qui ajoute un bouton permettant d'envoyer les données à la page `$page` qui sera alors exécutée. L'argument `$bouton_ok_str` est la chaîne de caractères affichée sur le bouton d'envoi des données. Le nom du formulaire `$id_form` doit être le même que le nom utilisé pour `debut_formulaire`.

Ces deux appels doivent encadrer tous les *composants* (voir ci-dessous) de ce formulaire, sans quoi les données ne pourront pas être envoyées.

Composants de formulaires

Un formulaire contient différents composants, permettant à l'utilisateur de saisir des données et de faire des choix.

Tout composant possède un `$nom` qui permet, lors du traitement des données saisies par l'utilisateur, de récupérer la valeur saisie dans le composant du nom donné (voir [GETref](#), section 3.4.2).

```
1 function : afficher_champ_texte( $nom , $texte [, $valeur ] )
2 |   $nom      : string
3 |   $texte    : string
4 |   $valeur   : string = ''
```

Ajoute au formulaire un champ de saisie de texte de nom `$nom`, précédé par la chaîne de caractères `$texte`. L'argument `$valeur` permet de donner une valeur par défaut qui sera affichée dans le champ de saisie.

```
1 function : afficher_champ_paragraphe( $nom , $texte [, $valeur , $rows
2 |   , $cols ] )
3 |   $nom      : string
4 |   $texte    : string
5 |   $valeur   : int      = ''
6 |   $rows     : int      = 4
7 |   $cols     : string   = 20
```

Comme `afficher_champ_texte`, mais permet de saisir plusieurs lignes de texte. Les arguments `$rows` et `$cols` permettent de modifier le nombre de lignes et de colonnes de la boîte de saisie.

```
1 function : afficher_champ_password( $nom , $texte [, $valeur ] )
2 |   $nom      : string
3 |   $texte    : string
4 |   $valeur   : string = ''
```

Comme `afficher_champ_texte` mais affiche des '*' à la place des caractères saisis au clavier.

```
1 function : afficher_checkbox( $nom , $texte [, $checked ] )
2 |   $nom      : string
3 |   $texte    : string
4 |   $checked  : bool   = false
```

Ajoute au formulaire une case à cocher seule, précédée du texte `$texte`. L'argument `$checked` définit si la case est cochée par défaut.

```
1 function : debut_groupe_checkbox( $nom , $texte )
2 |   $nom      : string
3 |   $texte    : string
```

```
1 function : afficher_checkbox_item( $nom , $texte [, $checked ] )
2 |   $nom      : string
3 |   $texte    : string
4 |   $checked  : bool   = false
```

```
1 function : fin_groupe_checkbox( [ $nom ] )
2 |   $nom      : string = ''
```

Ajoutent un groupe de cases à cocher au formulaire. Le groupe doit commencer par `debut_groupe_checkbox`, suivi de plusieurs appels à `afficher_checkbox_item`, et terminer par `debut_groupe_checkbox`. Chaque appel à `afficher_checkbox_item` ajoute une case à cocher au groupe.

Par exemple :

```
1 debut_groupe_checkbox('animaux', 'Quels animaux possédez-vous ?');
2   afficher_checkbox_item('chat', 'Chat');
3   afficher_checkbox_item('chien', 'Chien', true); // coché par défaut
4   afficher_checkbox_item('lapin', 'Lapin', true); // coché par défaut
5 fin_groupe_checkbox('animaux');
```

Pour savoir comment les valeurs peuvent être récupérées, voir 3.4.2.

```

1 function : debut_groupe_radio( $nom , $texte )
2   | $nom      : string
3   | $texte    : string

```

```

1 function : afficher_radio_item( $nom , $texte [, $checked ] )
2   | $nom      : string
3   | $texte    : string
4   | $checked  : bool   = false

```

```

1 function : fin_groupe_radio( [ $nom ] )
2   | $nom      : string = ''

```

Ajoutent un groupe de boutons radio au formulaire. Le groupe doit commencer par `debut_groupe_radio`, suivi de plusieurs appels à `afficher_radio_item`, et terminer par `debut_groupe_radio`. Chaque appel à `afficher_radio_item` ajoute un bouton radio au groupe.

Un seul bouton peut être sélectionné dans tout le groupe.

```

1 function : debut_select( $nom , $texte )
2   | $nom      : string
3   | $texte    : string

```

```

1 function : afficher_select_item( $texte [, $valeur ] )
2   | $texte    : string
3   | $valeur   : string = false

```

```

1 function : fin_select( [ $nom ] )
2   | $nom      : string = ''

```

Ajoutent une liste déroulante au formulaire. La création de la liste doit commencer par `debut_select`, suivi de un ou plusieurs appels à `afficher_select_item`, et terminer par `fin_select`. Chaque appel à `afficher_select_item` ajoute un élément à la liste.

L'argument `$valeur` permet de définir la valeur qui sera envoyée si cet élément est le choix de l'utilisateur. Par défaut cette valeur est la même que le texte affiché `$texte`.

```

1 function : ajouter_champ_cache( $nom , $valeur )
2   | $nom      : string
3   | $valeur   : string

```

Ajoute un champ caché au formulaire. Cela permet d'envoyer des données à travers le formulaire en même temps que les autres valeurs saisies par l'utilisateur, sans rien afficher.

3.4.2 Traitement des données de formulaire

Lorsque l'utilisateur clique sur le bouton 'Ok' d'un formulaire affiché à l'écran, cela demande au serveur PHP d'exécuter le fichier PHP donné en paramètres du formulaire. Ce fichier est (peut-être à nouveau) *entièrement* exécuté. PHP n'a pas de mémoire de la dernière exécution, donc les seules valeurs qui changent d'une exécution à l'autre d'un fichier PHP sont les valeurs reçues d'un formulaire.

Pour savoir si l'exécution courante du fichier PHP est due à l'envoi de données d'un formulaire, on peut tester le contenu de la variable `$id_formulaire`, qui est passée automatiquement en argument de la fonction `main` :

```

1 function main($id_formulaire) {
2     if($id_formulaire == 'nom_du_formulaire') {
3         // ... traitement des données du formulaire
4     } else {
5         // ... on n'est pas passé par le formulaire
6     }
7 }

```

La variable `$id_formulaire` contient le nom du formulaire qui a envoyé les données, ou `false` si on est arrivé sur la page courante sans passer par un formulaire. On peut donc utiliser cette variable pour savoir quelles instructions effectuer.

Récupération des données saisies

Pour récupérer une donnée saisie par l'utilisateur dans le formulaire, il faut utiliser la fonction :

```

1 function : GETref( $nom ) -> string
2 | $nom : string

```

qui renvoie la valeur de champ de formulaire de nom `$nom`. Cette valeur est soit `false`, soit une chaîne de caractères².

Voici un exemple de page PHP affichant un formulaire :

```

1 <?php
2
3 function afficher_formulaire() {
4     debut_formulaire('form_sport', 'traitement.php');
5     afficher_champ_texte('sport', 'Sport favori :');
6     afficher_champ_texte('heures', 'Nombre d\'heures de pratique
7     par semaine :');
8     fin_formulaire('form_sport');
9 }

```

2. Notez qu'en PHP, les valeurs 0, "" et `false` sont équivalentes vis-à-vis de l'opérateur `==` ; et qu'une chaîne de caractère non-vide est équivalente à `true` pour les tests, par exemple dans les `if` et `while`.

```

10 function main() {
11     afficher_formulaire();
12 }
13
14 ?>

```

Et voici la page qui réceptionne les données :

```

1 <?php
2
3 function traiter_formulaire() {
4
5     // Récupération des données saisies dans le formulaire
6     $sport = GETref('sport');
7     $heures = GETref('heures');
8
9     // Utilisation des données
10    printline('Votre sport est ' . $sport . ' que vous pratiquez ' .
11             $heures . ' heures par semaine');
12 }
13
14 function main($id_form) {
15
16     if($id_form == 'form_sport') {
17         // Si on est arrivé sur cette page à cause d'une pression
18         // sur le bouton Ok du formulaire 'form_sport'
19         traiter_formulaire();
20     } else {
21         // Sinon on est arrivé ici par un autre formulaire
22         // ou sans formulaire du tout
23         printline('Vous n\'êtes pas passés par le formulaire !');
24     }
25 }
26
27 ?>

```

Récupération des données d'une seule checkbox

La fonction `GETref` sur les checkbox renvoie le texte de la checkbox. Il est préférable d'utiliser la fonction :

```

1 function : GETcheckbox( $nom ) -> string
2 | $nom : string

```

qui renvoie `true` si la checkbox `$nom` a été cochée, `false` sinon.

Récupération des données d'un groupe checkbox

Pour récupérer les données d'un groupe checkbox, il faut aussi utiliser `GETref`, ce qui renvoie un tableau associatif, où chaque clef est une des cases **cochées** par l'utilisateur. Il n'y a donc pas de clef pour les cases non-cochées. Pour savoir si une clef existe dans le tableau, utiliser la fonction `array_key_exists` (voir A.2.6 ainsi qu'un exemple d'utilisation en section 3.5).

Auto-validation de formulaire

Il est tout à fait possible de faire traiter le formulaire par la page qui a envoyé les données. Dans ce cas, il faut bien avoir à l'esprit que les données ne seront traitées **qu'à la prochaine exécution du formulaire**.

Voir l'exemple complet en section 4.3.

Redirection de page

Si l'on souhaite rediriger l'utilisateur vers une autre page PHP, on peut utiliser la fonction :

```
1 function : redirection( $page )  
2 | $page : string
```

où `$page` est le nom du fichier PHP relativement au répertoire courant.

Attention, aucun affichage ne doit avoir été fait au moment où cette fonction est exécutée.

3.5 Sessions

Lorsque PHP passe d'une page à l'autre, il ne mémorise pas les valeurs des variables, il repart toujours de zéro. Pour pouvoir conserver des données d'une page à l'autre, on peut utiliser des champs cachés dans des formulaires, mais cela devient rapidement peu pratique quand le nombre de valeurs à mémoriser augmente.

Le mécanisme adapté pour ce type de mémorisation est celui des *sessions*.

La session est simplement un tableau associatif `$_SESSION[]`, où l'on peut associer des clefs à des valeurs à mémoriser.

Par exemple, pour mémoriser la valeur 12 pour la clef 'age', il suffit de faire :

```
1 $_SESSION['age'] = 12;
```

Pour savoir si une clef existe dans la session, on peut utiliser `array_key_exists`.

Voici le programme "compteur.php" qui permet de compter le nombre de fois où la page a été exécutée :

```

1 <?php
2
3 function main() {
4
5     // Si la clef 'compteur' existe déjà dans la mémoire
6     // de la session
7     if(array_key_exists('compteur', $_SESSION)) {
8         // alors récupérer sa valeur dans la variable $compteur
9         $compteur = $_SESSION['compteur'];
10    } else {
11        // La clef 'compteur' n'existe pas dans la session
12        // On donne donc une valeur initiale
13        $compteur = 0;
14    }
15
16    // Augmenter la valeur du compteur
17    $compteur++;
18
19    // Affichage
20    printline('Nombre d\'exécutions de la page : '. $compteur);
21    // Si on clique sur le lien suivant, cela exécute à nouveau la page
22    // et incrémente donc le compteur
23    afficher_lien_interne('Recharger la page', 'compteur.php');
24
25    // Mémoriser la nouvelle valeur du compteur
26    // pour l'exécution suivante.
27    // Si ce n'est la première fois que la page est exécutée,
28    // cela remplace l'ancienne valeur
29    $_SESSION['compteur'] = $compteur;
30 }
31
32 ?>

```

3.6 Modification du style

Il est possible d'ajouter une feuille de style avec la fonction :

```

1 function : ajouter_feuille_css( $feuille )
2 |   $feuille : string

```

où `$feuille` est le nom du fichier CSS relativement au répertoire courant.

Pour appliquer un style particulier à une section d'affichage, on peut utiliser les fonctions :

```

1 function : debut( $style )
2 |   $style : string

```

et :

```
1 function : fin([ $style ] )  
2 | $style : string = ''
```

où `$style` est une classe CSS (sans le "." la précédant) définie dans une feuille de style utilisée par le fichier PHP courant.

3.6.1 Style agro par défaut

Pour avoir une mise en page simple mais plus agréable qu'une page blanche, un style *agro* vous est fourni.

Pour vous en servir, il vous suffit d'inclure le fichier en tête de votre fichier PHP par la ligne :

```
1 include('../agro/agro-inc.php');
```

et de renommer votre fonction `main` en `agro_main` :

```
1 <?php  
2 include('../agro/agro-inc.php') ;  
3  
4 function agro_main($id_formulaire) {  
5     printline('Bienvenue sur mon site web !') ;  
6 }  
7 ?>
```

Le titre de la page peut-être modifié en redéfinissant, à l'extérieur de toute fonction, la variable globale `$SI_titre_page`.

Chapitre 4

PHP/MySQL

En PHP, il est très simple d'exécuter des requêtes SQL.

Dans un même script PHP, l'utilisation de MySQL se fait en 4 temps :

- Établissement d'une connexion générale à MySQL avec `mysql_connect`
- Sélection d'une base de données avec `mysql_select_db`
- Exécution des différentes requêtes et traitement des réponses
- Fermeture de la connexion avec `mysql_close`

4.1 Connexion à la base de données

La fonction :

```
1 function : mysql_connect( $server , $username , $password )  
2 | $server : string  
3 | $username : string  
4 | $password : string
```

permet d'établir une connexion au *serveur* MySQL. C'est ce serveur qui fait l'interface entre PHP et la base de données. Il est donc nécessaire d'établir cette connexion.

Lorsque la base de données est utilisée en *local*, c'est-à-dire que le même ordinateur sert à la fois de serveur PHP et de serveur MySQL (ce qui sera le cas pour vous), on utilise généralement cette fonction de cette manière :

```
1 mysql_connect('localhost', 'root', 'mysql') ;
```

ou parfois :

```
1 mysql_connect('localhost', 'root', '') ;
```

selon la configuration du serveur MySQL.

La fonction :

```
1 function : mysql_select_db( $database_name )
2 | $database_name : string
```

permet de sélectionner la base de données `$database_name`.

4.1.1 Exécution de requêtes SQL

La fonction :

```
1 function : mysql_query( $requete ) -> resource
2 | $requete : string
```

demande au serveur MySQL d'exécuter la requête SQL `$requete` et renvoie son résultat sous forme d'une *ressource*.

Cette fonction ne fait cependant pas de vérification sur la requête en cas d'erreur. Pour cela, une fonction de remplacement vous est fournie dans le paquetage AgroSIXPack :

```
1 function : mysql_query_debug( $requete ) -> resource
2 | $requete : string
```

qui fait la même chose que `mysql_query` mais affiche des messages d'erreur plus informatifs lorsque l'exécution de la requête n'a pas fonctionné.

Le paquetage AgroSIXPack fournit aussi la fonction :

```
1 function : mysql_query_affiche( $requete ) -> resource
2 | $requete : string
```

qui fait la même chose que `mysql_query_debug` mais affiche en plus à l'écran la chaîne de caractères correspondant à la requête à exécuter.

Ceci est très pratique pour visualiser rapidement d'où vient l'erreur, et permet aussi de tester la requête par un copier/coller dans phpMyAdmin.

Vous devez donc utiliser `mysql_query_affiche` pour tester vos requêtes, et une fois que vous êtes assurés qu'elles fonctionnent correctement, vous pouvez utiliser `mysql_query_debug` pour éviter l'affichage.

4.1.2 Requêtes de type SELECT

Pour les requêtes de type SELECT, il faut voir la ressource retournée comme une table résultat SQL, *y compris lorsque l'on sait qu'une seule valeur est retournée !*

Attention, ces ressources ne sont pas des tableaux PHP ; il faut utiliser des fonctions particulières pour en extraire le contenu.

Pour lire le contenu de cette ressource donc, il faut parcourir la table résultat ligne par ligne, grâce à la fonction :

```

1 function : mysql_fetch_assoc( $res ) -> array
2   | $res : resource

```

qui prend en argument une resource récupérée d'un appel précédent à `mysql_query`, et qui renvoie un tableau associatif PHP correspondant à la *prochaine* ligne de la table SQL résultat.

S'il n'y a pas de prochaine ligne dans le tableau, `mysql_fetch_assoc` renvoie `false`.

Les clefs ou index du tableau sont les noms des colonnes de la table résultat demandés dans le SELECT, et les valeurs associées sont celles qui correspondent dans la table résultat.

Exemple d'utilisation :

```

1 $res = mysql_query_debug('SELECT prenom, nom FROM personnes');
2 $ligne1 = mysql_fetch_assoc(res);
3 if($ligne1) {
4     printline('prenom = ' . $ligne1['prenom'] .
5               ' nom = ' . $ligne1['nom'] ) ;
6 }

```

Ici, si on ne passe pas dans le `if`, cela signifie que `$ligne1` est faux, ce qui signifie que la requête a renvoyé une table avec 0 ligne, donc aucun résultat. Sinon, l'appel à `mysql_fetch_assoc($res)` permet de récupérer la première ligne de la table résultat, que l'on demande ensuite d'afficher.

Notez que dans l'exemple précédent, on ne s'est intéressé qu'à la première ligne du résultat, et les autres lignes n'ont pas été traitées.

Pour traiter *toutes* les lignes de la même manière, on utilise généralement une boucle `while` :

```

1 $res = mysql_query_debug('SELECT prenom, nom FROM personnes');
2 while($ligne = mysql_fetch_assoc(res)) {
3     printline('prenom = ' . $ligne['prenom'] .
4               ' nom = ' . $ligne['nom'] ) ;
5 }

```

Cet exemple-ci affiche donc tous les résultats de la requête. Notez que cette fois-ci, l'affectation de `$ligne` est fait dans le test de la boucle `while`, ce qui a pour effet à la fois de modifier la valeur de `$ligne`, et de tester que cette nouvelle valeur n'est pas `false`, sinon on arrête la boucle car il n'y a plus de ligne à traiter.

4.1.3 Autres types requêtes

Pour les autres types de requêtes (UPDATE, INSERT INTO, DELETE, etc.), la resource renvoyée est `true` ou `false` selon que la requête a pu être exécutée ou non.

4.1.4 Échappement des apostrophes

Dans une requête SQL, les valeurs sont encadrées par des apostrophes simples :

```
1 SELECT * FROM personnes WHERE nom='Dupondt'
```

Mais si la valeur contient elle-même une apostrophe, il est nécessaire de *l'échapper*, c'est-à-dire de faire comprendre au langage qu'il ne s'agit pas de la fin de la chaîne de caractères, mais bien d'une caractères à l'intérieur de la chaîne, en faisant précéder l'apostrophe par un *antislash* `'\'` :

```
1 SELECT * FROM personnes WHERE nom='l\'Autre'
```

En PHP, de telles valeurs proviennent généralement d'un formulaire, c'est-à-dire qu'elles sont données par l'utilisateur. Il devient alors nécessaire d'utiliser une méthode générale pour échapper automatiquement les apostrophes. Pour cela, on utilise la fonction `mysql_real_escape_string` :

```
1 $nom = ...;
2 // la variable $nom contient une chaîne quelconque
3 $res = mysql_query_affiche('SELECT * FROM personnes WHERE nom=\'' .
4     mysql_real_escape_string($nom) . '\''');
5 ...
```

L'utilisation de cette fonction est primordiale, car elle empêche aussi un utilisateur malveillant de pouvoir faire des *injections SQL*, c'est-à-dire de pouvoir faire des requêtes non-autorisées sur votre base de données, ce qui pourrait, par exemple, lui donner accès aux mots de passes de tous les utilisateurs, ou lui permettre d'effacer entièrement la base.

Notez qu'il faut cependant échapper les apostrophes encadrant la valeur. Pour éviter cela, on peut aussi utiliser des apostrophes doubles pour les valeurs MySQL :

```
1 $nom = ...;
2 // la variable $nom contient une chaîne quelconque
3 $res = mysql_query_affiche('SELECT * FROM personnes WHERE nom="' .
4     mysql_real_escape_string($nom) . '"');
5 ...
```

4.2 Fermeture de la connexion à la base

Une fois que toutes les requêtes ont été traitées (et pas avant!), il faut fermer la connexion par un appel à la fonction :

```
1 function : mysql_close()
```

4.3 Exemple complet

En considérant que l'on a créé une base de données nommée "simpsons", possédant une table "personnes" ayant une colonne "prenom" et une colonne "nom", le fichier PHP suivant permet de demander un nom de famille à l'utilisateur et lui affiche en retour la liste des personnages possédant ce nom.

Contenu du fichier "ex-mysql.php" :

```

1 <?php
2
3 function afficher_formulaire() {
4
5     // Création du formulaire
6     debut_formulaire('form_nom', 'ex-mysql.php');
7
8     // Ajout d'un champ texte de nom 'nom'
9     afficher_champ_texte('nom', 'Nom : ');
10
11    fin_formulaire('form_nom');
12 }
13
14 function afficher_prenoms($nom) {
15     // Création de la requête à partir du texte saisi par l'utilisateur
16     // Utiliser mysql_real_escape_string !
17     $requete = 'SELECT prenom, nom FROM personnages WHERE nom LIKE \'' .
18         . mysql_real_escape_string($nom) . '%\'';
19
20     // Afficher la requête à l'écran, puis l'exécuter et
21     // récupérer la table SQL résultat
22     $res = mysql_query_affiche($requete);
23
24     // Afficher le résultats dans une liste
25     debut_liste();
26
27     // parcourir la table SQL résultat, ligne par ligne :
28     while($ligne = mysql_fetch_assoc($res)) {
29         afficher_liste_item($ligne['prenom'] . ' ' . $ligne['nom']);
30     }
31
32     fin_liste();
33 }
34
35 function main($id_formulaire) {
36
37     // Attention : cette fonction n'est pas "bloquante",
38     // Elle n'attend pas que l'utilisateur ait entré des valeurs
39     // pour pouvoir continuer le programme.
40     afficher_formulaire();
41     // À la place, le programme *affiche* le formulaire puis
42     // passe à ce qui suit.

```

```
43 // Il faudra attendre la *prochaine* exécution de ce fichier pour
44 // traiter les données de l'utilisateur.
45
46 // Traitement des données reçues :
47 if($id_formulaire == 'form_nom') {
48
49     // Récupération du texte inscrit dans le champ 'nom'
50     // par l'utilisateur à l'exécution *précédente*
51     // de ce fichier
52     $nom = GETref('nom');
53
54     // Ouverture de la connexion au serveur MySQL
55     mysql_connect('localhost', 'root', 'mysql');
56     // Selection de la base de données
57     mysql_select_db('simpsons');
58
59     // Appel à la fonction d'affichage des prénoms
60     afficher_prenoms($nom);
61
62     // fermeture de la connexion à la base
63     mysql_close();
64 }
65
66 }
67 ?>
```

4.4 Pour aller plus loin

Pour des informations plus complètes sur les fonctions MySQL en PHP, vous pouvez consulter la page web : <http://www.php.net/manual/fr/ref.mysql.php>.

Chapitre 5

Travaux Dirigés PHP

TD 1 : Variables et fonctions

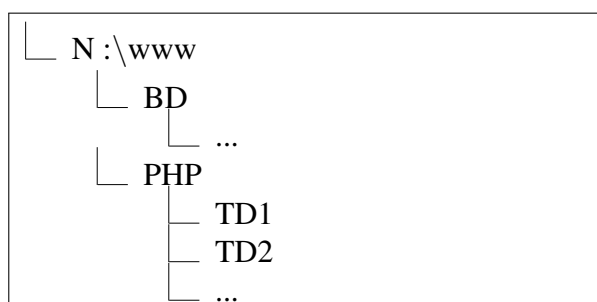
Pour effectuer ces exercices, vous devez avoir lu les pages 8 à 34 avant de venir en TD.

En premier lieu, il vous faut récupérer l'archive zip des exercices de PHP. Vous la trouverez à l'adresse suivante :

<http://www.agroparistech.fr/Systeme-d-Information.html>

Il est important de respecter à la lettre la marche à suivre que voici : Déplacez l'archive dans votre répertoire N : \www. Cliquez dessus avec le bouton droit de la souris et choisissez "Décompresser ici".

Ceci doit vous créer l'arborescence suivante :



Chaque TD a donc son propre répertoire, et pour aujourd'hui tous les exercices se trouvent dans TD1.

Exercice 1 : Premiers pas

Votre premier programme en PHP.

Commencez par lancer EasyPHP à partir du menu démarrer. Ouvrez ensuite Notepad++ à partir du menu démarrer. Puis, dans Notepad++, ouvrez le fichier :

N : \www \PHP \TD1 \exo1.php

Complétez la fonction `main` avec le code PHP suivant :

```
1 printline('Bonjour le monde !');
```

N'oubliez pas de sauvegarder votre fichier, sinon les changements ne seront pas pris en compte.

Tapez ensuite l'adresse suivante dans votre navigateur :

<http://localhost/www/PHP/AgroSIXPack.php>

puis choisissez le fichier "exo1.php" dans le répertoire "TD1". Ceci a pour effet d'*exécuter* le fichier "exo1.php" dans le navigateur.

Remarque : Si par la suite vous faites des changements au fichier "exo1.php", il vous suffira dans le navigateur de cliquer sur "Actualiser" ou "Rafraichir la page" ou bien d'ap-

puyer sur la touche F5 pour recharger le fichier sans avoir à retaper l'adresse URL.

Exercice 2 : Variables

Ouvrez le fichier N : \www\PHP\TD1\exo2.php dans Notepad++.
Complétez-le de manière à afficher :

```
Valeur de var1 : 5  
Valeur de var2 : 12
```

sous la contrainte que les seules opérations que vous avez le droit de faire sont :

- Additionner, soustraire, multiplier, diviser les variables déjà données,
- Modifier la valeur des variables données.

De plus, vous n'avez pas le droit d'utiliser d'autres nombres que ceux déjà donnés et vous ne pouvez effectuer que 3 opérations arithmétiques (+, -, *, /).

Exercice 3 : Correction d'erreurs

Exécutez le fichier N : \www\PHP\TD1\exo3.php dans le navigateur.

Chaque fonction (hormis la fonction `main`) comporte une et une seule erreur. À son exécution, PHP vous indique que ce fichier est incorrect, et vous fournit un message d'avertissement.

Question 1 :

Pour chaque erreur que vous rencontrez, lisez-bien le message qui vous est donné. PHP vous y indique notamment le numéro de la ligne de votre fichier où se trouve l'erreur. L'erreur réelle se trouve généralement soit sur cette ligne, soit sur l'instruction la précédant.

Corrigez les 3 erreurs du fichier et vérifiez que son exécution se passe correctement.

Les messages sont en anglais, mais que cela ne vous empêche pas d'essayer de les comprendre. Si vous apprenez à comprendre les messages d'erreurs, cela vous fera gagner de nombreuses minutes plus tard !

Remarque : Vous n'avez que 3 caractères à ajouter et un seul à modifier pour que le fichier soit correct. Vous ne devez évidemment pas supprimer d'instructions...

Exercice 4 : Fonction de concaténation de chaînes de caractères

Question 1 :

Dans le fichier N : \www\PHP\TD1\exo4.php, écrivez la fonction `entourer` dont voici la *signature* (voir page 86) :

```

1 function : entourer( $chaine1 , $chaine2 ) -> string
2 |   $chaine1 : string
3 |   $chaine2 : string

```

Qui prend en argument 2 chaînes de caractères et qui retourne la chaîne `$chaine1` entourée à gauche et à droite par la chaîne `$chaine2`.

Une fois la fonction écrite, exécutez le fichier "exo4.php" pour vérifier qu'il est correct. Vous devez obtenir exactement :

```
Voici le dilemme : être ou ne pas être
```

Remarque : Vous ne devez pas modifier la fonction principale.

Exercice 5 : Abstraction

Ouvrez le fichier N : \www\PHP\TD1\exo5.php.

Question 1 :

Modifiez la fonction `main` pour lui faire afficher le résultat suivant :

```

Je pense
Tu penses
Il/elle/on pense
Nous pensons
Vous pensez
Ils/elles pensent

```

Trouvez un autre exemple avec un second verbe du premier groupe, et appliquez la méthode de l'abstraction (cf. 2.7.6) sur ces deux exemples pour obtenir au final une fonction suffisamment générale pour traiter différents verbes du premier groupe.

Exercice 6 : Variables locales

Ouvrez le fichier N : \www\PHP\TD1\var-locales.php.

Les comportements que vous allez observer sur les variables sont des notions très importantes de programmation. Ils sont propres à de nombreux langages, il est important de bien les comprendre.

Question 1 :

Dans la fonction `main`, après l'affectation des variables, ajoutez l'affichage de leur valeur par :

```
1 printline ('a = ' . $a . ' b = ' . $b) ;
```

Et exécutez ce fichier pour visualiser le résultat.

Question 2 :

Juste en dessous de ce que vous venez d'écrire, ajoutez l'appel suivant à la fonction `fun1` :

```
1 fun1 ($a) ;
```

Exécutez le fichier et vérifiez si ce qui est affiché correspond bien à votre intuition.

Question 3 :

Ajoutez encore en-dessous un nouvel affichage :

```
1 printline ('a = ' . $a . ' b = ' . $b) ;
```

À nouveau, vérifiez que le résultat est bien conforme à votre intuition. Consultez la section 2.7.4 dans le cas contraire.

Question 4 :

Ajoutez enfin :

```
1 fun1 ($b) ;  
2  
3 printline ('a = ' . $a . ' b = ' . $b) ;
```

et vérifiez une dernière fois que vous avez bien compris comment fonctionnent les variables locales.

Exercice 7 : Multilinguisme

Ouvrez le fichier "langues.php" (et non "langues-form.php") dans Notepad++ et exécutez-le dans le navigateur.

Vous êtes automatiquement redirigé vers un formulaire de choix de langues. Lorsque vous appuyez sur le bouton "Ok", ceci a pour effet d'exécuter le fichier "langues.php" en lui envoyant des valeurs, que vous pouvez récupérer grâce à `GETcheckbox` dans la fonction `traiter_formulaire`, qui sera exécutée automatiquement (car appelée dans la fonction `main`).

Vous n'avez pas besoin de modifier la fonction `main`.

Question 1 :

Sur le modèle qui vous est donné, dans la fonction `compter_langues`, récupérez les valeurs des choix de l'utilisateur pour les autres langues, et affichez les valeurs reçues pour vérifier qu'elles sont correctes.

Remarque : En PHP, la chaîne de caractères vide "", le 0 et `false` sont semblables, de même que 1 et `true` sont semblables.

Question 2 :

Modifiez la fonction `compter_langues` pour compter le nombre de langues parlées par l'utilisateur.

Dans la fonction `traiter_formulaire`, faites un appel à la fonction `compter_langues` et récupérez la valeur de retour dans une variable `$nb_langues`. Ajoutez-y enfin un affichage, par exemple :

```
Vous parlez 2 langue(s).
```

Question 3 :

Modifiez la fonction `traiter_formulaire` pour afficher l'une des phrases suivantes en fonction du nombre de langues parlées : "Vous ne parlez aucune langue, j'ai du mal à y croire." ou bien "Vous ne savez parler qu'une seule langue, c'est peu...", ou bien "Vous êtes dans la moyenne." (pour 2 ou 3 langues parlées), ou bien "Impressionnant." (pour 4 langues parlées).

Question 4 :

Supprimez les affichages dans la fonction `compter_langues` : ils n'étaient destinés qu'à titre informatif et les fonctions qui retournent des valeurs ne doivent en général pas faire d'affichage.

Modifiez le code écrit à la question précédente de manière à n'utiliser qu'un seul et unique `printline`, qui doit être dans votre fonction `traiter_formulaire`.

TD 2 : Boucles et tableaux

Pour effectuer ces exercices, vous devez avoir lu les pages 35 à 42 avant de venir en TD.

Tous les exercices de ce TD se trouvent dans votre répertoire N : \www\PHP\TD2.

Remarque : Dans ce TD, nous voyons les boucles, vous ne devez donc pas énumérer les différents cas possibles un par un ; il faut chercher à faire des solutions générales, qui fonctionnent quelque soit le nombre de traitements individuels à faire.

Exercice 1 : 99 Bouteilles de bière

La chanson "99 bottles of beer" est un chant traditionnel américain. Nous allons le programmer lorsque le nombre de bouteilles initial est donné par l'utilisateur.

Question 1 :

Modifiez la fonction `traiter_formulaire` dans le fichier "bouteilles.php" pour lui faire afficher le résultat suivant, ici dans le cas où le nombre de bouteilles entré est 3 :

```
3 bottles of beer on the wall, 3 bottles of beer.  
Take one down, pass it around, 2 bottles of beer on the wall.  
  
2 bottles of beer on the wall, 2 bottles of beer.  
Take one down, pass it around, 1 bottles of beer on the wall.  
  
1 bottles of beer on the wall, 1 bottles of beer.  
Take one down, pass it around, 0 bottles of beer on the wall.  
  
No more bottles of beer on the wall, got to the store and buy some more.
```

Remarque : Il n'est pas nécessaire de traiter les cas particuliers des singuliers pour 0 et 1 (mais vous pouvez le faire si cela vous trouble...).

Exercice 2 : Triangle

Ouvrez le fichier "triangle.php".

Dans cet exercice, vous devez dessiner un triangle dont la longueur de la base est donnée par l'utilisateur dans le formulaire.

Par exemple, pour une base de taille 6, vous devez obtenir le résultat suivant :

```
*
**
***
****
*****
*****
```

Appliquez la méthode *diviser pour résoudre* (cf. section 2.7.7) pour résoudre ce problème.

Votre code correspondant à la création du triangle ne doit pas se trouver dans la fonction `traiter_formulaire`. Cette fonction doit seulement récupérer un triangle sous forme de chaîne de caractères par un appel à votre fonction et l'afficher à l'écran (donc vos fonctions additionnelles ne doivent pas faire d'affichage).

Exercice 3 : Jeux de mots

Ouvrez le fichier "phrases.php". L'objectif de cet exercice est de créer aléatoirement des phrases relativement simples.

Question 1 :

Créez la fonction `main`. Cette fonction ne devra contenir que des *appels de test* aux fonctions que vous allez définir. Votre code correspondant aux énoncés des questions devra être mis dans des fonctions différentes.

Ajoutez un affichage de test tel que `printline('Dans le main')` ; pour vérifier que votre programme s'exécute correctement. Retirez-le (ou mettez-le en commentaire) une fois le test passé.

Question 2 : Création des tableaux

Créez une fonction :

```
1 function : afficher_phrase()
```

qui ne prend aucun argument.

Dans cette fonction, créez 2 tableaux qui contiendront des chaînes de caractères :

Créez un premier tableau nommé `$objets` contenant 3 prénoms ainsi que 3 objets avec leur article ('une clef', 'le mouton', etc.).

Créez un tableau nommé `$verbes` contenant 4 verbes transitifs directs, conjugués à la 3ème personne du singulier (par exemple 'pousse')¹.

1. Vous pouvez utiliser des verbes transitifs indirects à condition de les faire suivre d'une préposition, par exemple 'pose sur'.

Rappel : Un verbe transitif direct est un verbe qui accepte un complément d'objet direct, comme 'prendre', 'déraciner', etc.

Exécutez votre programme pour vérifier que vous n'avez pas fait d'erreur de syntaxe.

Question 3 : Choix d'un élément

Faites une fonction :

```
1 function : choix_element( $tableau ) -> string
2 | $tableau : array(string)
```

qui prend en paramètre un tableau de chaînes de caractères et qui renvoie un élément pris au hasard dans ce tableau.

Vous utiliserez la fonction `rand` :

```
1 function : rand( $min , $max ) -> int
2 | $min : int
3 | $max : int
```

qui retourne un nombre entre les valeurs `$min` et `$max` incluses.

Rappel : le premier indice d'un tableau est le 0.

Rappel : la fonction `count` permet de connaître le nombre d'éléments d'un tableau.

Faites quelques tests avec cette fonction et les tableaux précédents pour vérifier qu'elle fonctionne correctement.

Question 4 :

Faites une fonction :

```
1 function : creer_phrase( $tab_objets , $tab_verbes ) -> string
2 | $tab_objets : array(string)
3 | $tab_verbes : array(string)
```

qui prend en argument un tableau d'objets et un tableau de verbes conjugués et qui, en utilisant la fonction de la question précédente, renvoie une phrase créée aléatoirement.

Une phrase est composée d'un sujet, d'un verbe et d'un complément d'objet. Par exemple :

```
Pierre questionne un canapé
```

Remarque : `creer_phrase` ne doit rien afficher, mais doit retourner une valeur.

Dans la fonction `afficher_phrase`, faites un appel à la fonction `creer_phrase` et faites-lui afficher la phrase ainsi obtenue.

Question 5 : Compositions

Écrivez une fonction :

```
1 function : creer_proposition( $tab_objets , $tab_verbes ) -> string
2 |   $tab_objets : array(string)
3 |   $tab_verbes : array(string)
```

qui prend en argument un tableau d'objets et un tableau de verbes conjugués et qui renvoie une proposition subordonnée aléatoire commençant par ' qui ...'. Par exemple :

```
qui pose un mouton
```

Modifiez la fonction `créer_phrase` qui doit maintenant appeler `creer_proposition` pour renvoyer une phrase comportant un nombre aléatoire de propositions compris entre 0 et 4.

Remarque : Ici, vous devez utiliser une boucle.

Modifiez la fonction `creer_phrase` précédente pour lui faire écrire une phrase composée d'un nombre aléatoire (entre 1 et 5) de propositions adjointes par la préposition "qui". Par exemple :

```
Pierre questionne un canapé qui pousse Jacques qui tapotte un océan
```

Exercice 4 : Vente en ligne

Ouvrez le fichier "articles.php".

Dans le formulaire de "vente en ligne", l'utilisateur doit saisir 5 articles et leur prix. Dans votre fichier "articles.php", dans la fonction `traiter_formulaire`, vous récupérez deux variables `$articles` et `$prix`, qui sont des tableaux scalaires.

Question 1 :

Dans la fonction `traiter_formulaire`, utilisez la fonction `print_r` pour afficher le contenu des deux tableaux.

Question 2 :

Faites une fonction :

```
1 function : prix_total( $tab_prix ) -> real
2 | $tab_prix : array(real)
```

qui retourne le total des prix contenus dans le tableau `$tab_prix` passé en argument.

Question 3 :

Écrivez une fonction :

```
1 function : afficher_ticket( $tab_articles , $tab_prix )
2 | $tab_articles : array(string)
3 | $tab_prix      : array(real)
```

permettant de récapituler les informations saisies sous une forme plus agréable que le `print_r` qui ne fait qu'un affichage très sommaire d'un tableau. Affichez en dessous le prix total calculé.

Voici un exemple d'une partie de l'affichage de la table :

```
Article : Prix
-----
Pommes : 1.20
Yaourts : 2.49
Chocolat : 48.15
...
-----
Total : 125.77
```

Remplacez vos utilisations du `print_r` par un appel à la fonction `afficher_ticket`.

Remarque : Ne cherchez pas à aligner les ":".

Remarque : Le total affiché doit évidemment être un appel à la fonction `prix_total` définie plus haut.

Question 4 :

Faites une fonction :

```
1 function : article_plus_cher( $tab_prix ) -> string
2 | $tab_prix : array(real)
```

qui retourne *l'indice* de l'article le plus cher que vous avez acheté.

Modifiez la fonction `afficher_ticket` pour lui faire afficher aussi l'article le plus cher avec son prix :

```
Article : Prix
```

```
-----
```

```
Pommes : 1.20
```

```
Yaourts : 2.49
```

```
Chocolat : 48.15
```

```
...
```

```
-----
```

```
Total : 125.77
```

```
Article le plus cher :
```

```
Chocolat : 48.15
```

TD 3 : Tableaux et formulaires

Pour effectuer ces exercices, vous devez avoir lu les pages 42 à 55 avant de venir en TD.

Exercice 1 : Formulaires

Nous allons commencer par créer un formulaire simple pour comprendre comment les informations sont passées d'une page PHP à l'autre.

Question 1 :

Dans le fichier "exo1-formulaire.php", créez la fonction `main`, insérez-y un affichage quelconque pour vérifier que votre programme minimaliste peut-être exécuté.

Question 2 :

Créez une fonction :

```
1 function : afficher_formulaire()
```

qui affiche un formulaire avec un champ texte de nom 'prenom' demandant le prénom de l'utilisateur.

Exécutez le fichier, puis validez le formulaire. Ceci vous amène sur la page "exo1-traitement.php" et vous affiche un message d'erreur, puisque nous n'avons pas encore rempli ce fichier.

Question 3 :

Dans le fichier "exo1-traitement.php", créez la fonction `main` vide :

```
1 function : main( $id_formulaire )  
2 | $id_formulaire : string
```

Elle doit prendre en argument la variable `$id_formulaire`, qui représente le nom du formulaire qui a envoyé les données (entraînant l'exécution de ce fichier PHP), ou bien la chaîne vide "" si l'exécution de ce fichier n'est pas due à l'envoi de données par un formulaire.

Question 4 :

Créez ensuite la fonction :

```
1 function : traiter_formulaire()
```

et ajoutez-y un affichage de test dans la fonction `traiter_formulaire`, tel que :

```
1 printline('Dans traiter_formulaire()');
```

Faites un appel à cette fonction dans le `main`.

Exécutez le fichier "exo1-traitement.php". Vous devez voir l'affichage de votre texte de test.

Exécutez aussi le fichier "exo1-formulaire.php". Validez le formulaire : vous devez arriver sur la page "exo1-traitement.php" qui doit alors afficher aussi le message de test.

Question 5 :

Dans le fichier "exo1-traitement.php", modifiez la fonction `main` pour :

- Afficher le message de test uniquement si l'on valide le formulaire de la page "exo1-formulaire.php",
- Afficher un message tel que 'Désolé, vous devez passer par le formulaire ' si l'utilisateur a exécuté le fichier "exo1-traitement.php" directement.

Vous pouvez ajouter un lien interne (avec `afficher_lien_interne`) pour rediriger l'utilisateur vers la page du formulaire. Alternativement, vous pouvez utiliser la fonction `redirection` pour effectuer une redirection automatique.

Question 6 :

Nous allons maintenant récupérer les données que l'utilisateur saisit dans le formulaire. Pour cela, modifiez la fonction `traiter_formulaire` pour récupérer le prénom saisi par l'utilisateur grâce à un appel à la fonction `GETref`.

Affichez le récapitulatif des informations saisies :

```
Prénom : Albert.
```

Question 7 :

Dans le formulaire, ajoutez un *groupe radio* pour demander le sexe H/F de l'utilisateur.

Dans le traitement du formulaire, ajoutez la récupération des données du formulaire pour ce nouveau champ. Modifiez le récapitulatif des informations saisies par l'utilisateur en conséquence.

Question 8 :

Dans le formulaire, ajoutez une *liste de choix* (composant *select*) du style musical, dans laquelle vous ajoutez un certain nombre de styles musicaux selon vos goûts.

Modifiez le traitement du formulaire en conséquence, ainsi que le récapitulatif des informations saisies.

Question 9 :

Ajoutez enfin une *checkbox* demandant à l'utilisateur s'il est lui-même musicien, et modifiez le traitement du formulaire en conséquence.

Exercice 2 : Formulaire auto-validé

Nous voulons faire un formulaire demandant à l'utilisateur une taille de séquence ADN à générer aléatoirement, ainsi qu'une sous-séquence à chercher dans cette séquence.

Question 1 :

Dans le fichier "auto-form.php", créez la fonction `main` :

```
1 function : main( $id_formulaire )  
2 | $id_formulaire : string
```

Elle doit prendre en argument une variable `$id_formulaire`.

Question 2 :

Créez une fonction :

```
1 function : generer_ADN( $taille ) -> array(string)  
2 | $taille : int
```

qui renvoie une séquence ADN de taille `$taille` sous forme d'un tableau de caractères (chaque case doit être un acide aminé 'A', 'C', 'T' ou 'G').

Testez votre fonction pour vérifier qu'elle fonctionne correctement.

Question 3 :

Créez une fonction :

```
1 function : chercher_TTT( $adn ) -> bool  
2 | $adn : array(string)
```

qui cherche la chaîne 'TTT' dans l'ADN passé en argument, et renvoie `true` si elle le trouve, `false` sinon.

Testez votre fonction pour vérifier qu'elle fonctionne correctement.

Question 4 :

Créez une fonction

```
1 function : afficher_formulaire()
```

ainsi qu'une fonction

```
1 function : traiter_formulaire()
```

Le formulaire doit envoyer ses données à la même page "auto-form.php". Il vous faut alors traiter correctement les différents cas dans la fonction `main`, qui doit prendre en argument la variable `$id_formulaire`, selon la manière dont la page est exécutée (soit parce que l'utilisateur a appuyé sur 'Envoyer' dans le formulaire, auquel cas la variable `$id_formulaire` a comme valeur le nom de votre formulaire, soit parce que la page a été appelée directement sans passer par un formulaire, auquel cas la variable `$id_formulaire` a comme valeur la chaîne vide).

Question 5 :

Remplissez la fonction `traiter_formulaire` pour générer la séquence ADN en fonction de la taille saisie par l'utilisateur.

Question 6 : Validation des données

Dans le cas où les données saisies par l'utilisateur ne sont pas correctes, vous devez afficher un message d'erreur l'invitant à saisir à nouveau ses données, et donc afficher aussi le formulaire. Si toutes les données sont correctes, le formulaire ne doit pas être affiché.

Indication : votre fonction `traiter_formulaire` devrait maintenant retourner un booléen qui vaut `true` si les informations saisies sont correctes, ou `false` sinon. Ce que l'on doit faire ensuite dans le cas `false` doit alors se trouver en dehors de la fonction `traiter_formulaire`.

Vous pouvez utiliser la fonction :

```
1 function : is_numeric( $v ) -> bool
2 | $v : any
```

qui renvoie `true` si `$v` est un nombre.

TD 4 : Sessions et tableaux associatifs

Pour effectuer ces exercices, vous devez avoir lu les pages 55 à 57 avant de venir en TD.

Exercice 1 : Compteur

Copiez le code du programme "compteur" (section 3.5) dans le fichier "compteur.php" et testez-le.

Question 1 :

Modifiez le programme pour que lorsque le compteur atteint 10, on appelle la fonction `session_unset()`, **juste avant l'accolade fermante du `main`**.

Exécutez plusieurs fois la page. Quel l'effet de cette modification ?

Exercice 2 : Labyrinthe

L'objectif de cet exercice est de créer un mini-jeu où l'on doit amener le joueur vers un but à travers un labyrinthe.

N'hésitez pas à ajouter des fonctions intermédiaires selon vos besoins ! Et même à modifier votre code après coup pour le rendre plus simple en ajoutant des fonctions.

Question 1 :

Ouvrez le fichier "labyrinthe.php". Ajoutez-y une fonction `main`, prenant en argument un `$id_formulaire`.

Question 2 :

Une matrice est un tableau PHP où chaque case contient elle-même un tableau PHP. Dans le `main`, définissez la matrice `$mat1` suivante :

```
1 $mat1 = array( array('a', 'b', 'c', 'd'),  
2               array('1', '2', '3', '4'),  
3               array('w', 'x', 'y', 'z') );
```

Créez une fonction :

```
1 function : afficher_labyrinthe( $matrice )  
2 |   $matrice : array(array(string))
```

qui prend en argument une matrice et l'affiche à l'écran. Utilisez pour cela les fonctions d'affichage de table PHP (voir les fonctions `debut_table` et autres). Pour l'argument

`$nom` de `debut_table`, donnez le nom 'laby' à la table : ceci permettra d'avoir un affichage automatiquement correctement formaté.

Testez dans le `main` l'affichage de la matrice `$mat1`.

Vérifiez bien que les lignes et les colonnes ne sont pas transposées.

Question 3 :

Créez une fonction :

```
1 function : creer_matrice( $nb_lignes , $nb_colonnes ) -> array(array(
   string))
2 |   $nb_lignes    : int
3 |   $nb_colonnes  : int
```

qui crée un tableau de `$nb_lignes` cases, dont chaque case contient elle-même un tableau de `$nb_colonnes` cases. Chaque case doit contenir le symbole 'X' (nous le changerons plus tard).

Remarque : Vous aurez besoin de 2 boucles imbriquées, ou bien vous pouvez décomposer le problème en faisant d'abord une sous-fonction affichant uniquement une ligne.

Dans le `main`, créez une matrice de 10 lignes et 5 colonnes avec la fonction `creer_matrice`, et affichez là avec la fonction `afficher_labyrinthe`. (Vous pouvez maintenant supprimer l'affichage de `$mat1`.)

Question 4 :

Modifiez la fonction `creer_matrice` pour que seules les cases du bord aient la valeur 'X' (pour un mur), les autres devant avoir la valeur ''.

Testez votre fonction.

Question 5 :

Pour éviter de créer une nouvelle matrice à chaque exécution de la page PHP, il nous faut la sauvegarder dans une *session* (voir la section 3.5).

Dans le `main`, ajoutez une nouvelle matrice dans la session si aucune n'y existe déjà ; puis récupérez la matrice stockée dans la session pour l'afficher.

Exécutez plusieurs fois la page PHP pour vérifier que la matrice n'est bien créée qu'une seule fois (dans le navigateur, appuyez plusieurs fois sur F5 ou rafraichissez la page).

Question 6 :

Nous allons maintenant avoir besoin de boutons d'action. Un bouton d'action est simplement un formulaire sans autre composant que le bouton d'envoi des données.

Comme nous allons avoir besoin de plusieurs boutons, commencez par créer une fonction :

```
1 function : afficher_bouton( $nom , $texte )
2 |   $nom    : string
3 |   $texte  : string
```

qui affiche un formulaire de nom `$nom` et de texte de bouton d'envoi `$texte` (référez-vous à la documentation de `afficher_formulaire` si nécessaire).

Faites maintenant une fonction :

```
1 function : afficher_boutons ()
```

qui affiche un bouton de nom `'recommencer'` et de texte `'Recommencer'`.

Enfin, dans le `main`, si `$id_bouton` a la valeur `'recommencer'`, remplacez la matrice stockée en session par une autre matrice nouvellement créée.

Dorénavant, si vous cliquez sur le bouton, cela doit afficher une nouvelle matrice, mais si vous rafraîchissez la page, la matrice ne doit pas changer.

Question 7 :

Nous allons maintenant ajouter un joueur qui pourra se déplacer dans le labyrinthe.

Le joueur est défini par sa position (ligne, colonne), qu'il faut sauver dans la session.

Faites les modifications nécessaires dans votre programme pour :

- Créer les positions initiales du joueur (par exemple en 1, 1) si elles n'existent pas dans la session et les y sauver,
- Modifier l'affichage de la matrice pour afficher le joueur au bon endroit.

Le joueur sera représenté par `'O'`.

Question 8 :

Ajoutez un bouton 'Droite' (de nom `'droite'`) dans la fonction `afficher_boutons`.

Faites ensuite une fonction :

```
1 function : bouger( $direction , $ligne , $colonne , $matrice )
2 |   $direction : string
3 |   $ligne    : int
4 |   $colonne  : int
5 |   $matrice  : array(array(string))
```

qui prend en argument une direction (`'droite'`, `'gauche'`, `'haut'` ou `'bas'`), la position courante du joueur et la matrice, et qui modifie **dans la session** la position du joueur en fonction de la direction donnée.

Dans un premier temps, ne faites que la direction `'droite'`, puis une fois que cela fonctionne, ajoutez les autres directions (et les boutons associés). Vous aurez à modifier certaines autres fonctions, notamment le `main`.

Question 9 :

Ajoutez aléatoirement des murs au milieu du labyrinthe. Un clic sur 'Recommencer' doit remettre le joueur en position initial (1, 1) et modifier la disposition des murs.

Question 10 :

Modifiez votre programme pour ajouter une case spéciale '*' dans la matrice. Lorsque le joueur atteint cette case, affichez un message de victoire.

Question 11 : Bonus

Ajoutez un adversaire automatique qui se dirige vers le joueur. S'il l'attrape avant que le joueur atteigne l'étoile, le joueur a perdu.

TD 5 : PHP/MySQL

Pour effectuer ces exercices, vous devez avoir lu les pages 58 à 63 avant de venir en TD.

Exercice 1 : Les Simpson

Importez la base de données "simpsons.sql".

Il y a 4 pages PHP, numérotées par ordre d'utilisation. Chaque page doit renvoyer à la suivante et la dernière doit afficher un lien de retour vers la première.

- Sur la première page doit être affichée une liste des noms de familles existants (sans doublon) dans la base.
- La validation de ce formulaire renvoie à "page2.php" qui doit proposer une liste des prénoms possibles pour le nom choisi.
- Ce qui renvoie à la page 3, où l'on demande d'écrire un commentaire sur le personnage choisi.
- La validation de ce dernier formulaire renvoie alors à la page 4, où tous les commentaires du personnage sélectionné doivent être affichés dans une *liste d'items*.

Question 1 :

Réalisez les pages les unes après les autres.

Vous devrez créer la table 'commentaires' avec la structure qui convient.

Question 2 :

Veillez à ce que si l'utilisateur exécute une des pages directement (sans passer par un formulaire), il soit redirigé vers "page1.php".

Question 3 :

Dans le même répertoire, créez un nouveau fichier "tout.php" dans lequel vous devez faire la même chose que ci-dessus, mais dans un seul et unique fichier.

Identifiez enfin les morceaux de code qui se ressemblent beaucoup de manière et créez des abstractions (fonctions, boucles, etc.) permettant de diminuer au maximum les redondances de code.

Choisissez des noms clairs pour vos fonctions et documentez-les (types des arguments et de la valeur de retour, petite phrase d'explication).

TD 6 : PHP/MySQL, Modélisation

Exercice 1 : Réseau social

Le but de cet exercice est de créer un mini réseau social, où chaque personne inscrite peut voir, ajouter, supprimer des amis, où l'on peut aussi ajouter et supprimer des inscrits.

Question 1 : Base de données

Commencez par créer la base de données permettant de modéliser le problème.

Réfléchissez bien à la modélisation de manière à ce que votre base de données **ne contienne aucune redondance d'information**.

Puis ajoutez quelques enregistrements pour pouvoir tester des requêtes.

Question 2 :

Commencez par faire une page permettant de voir les amis d'une personne choisie dans un formulaire.

Question 3 :

Faites ensuite les autres pages permettant de :

- Ajouter une nouvelle personne dans la base (un nouvel inscrit),
- Ajouter, à une personne inscrite, un ami parmi les personnes inscrites qui ne sont pas encore ses amis,
- Supprimer un ami (pas la personne inscrite, mais le lien d'amitié),
- Supprimer une personne (ce qui doit entraîner la suppression de tous ses amis !)

Veillez à bien gérer les cas particuliers et les erreurs possibles de l'utilisateur (ami inexistant ? Personne non-inscrite ? etc.).

Question 4 : Connexion

On aimerait maintenant pouvoir se "connecter" au site de manière à ce que toutes les requêtes soient relatives à la personne connectée, sans qu'elle ait besoin d'entrer son nom à chaque fois. Seule la personne connectée peut s'ajouter ou supprimer des amis, ou supprimer son compte.

Modifiez votre programme en conséquence. Vous aurez notamment besoin pour cela de mémoriser la personne connectée dans une session.

Annexe A

PHP : Addendum

A.1 Signatures de fonctions

Une signature de fonction permet de d'écrire les formats (types) d'entrée et sortie de la fonction, ce qui donne une information précieuse sur ce que fait la fonction.

Il ne s'agit pas de code PHP, c'est uniquement une description.

Prenons l'exemple suivant :

```
1 function : creer_cercle( $x , $y , $ray [, $epais , $couleur ] ) -> cercle
2 | $x      : int
3 | $y      : int
4 | $ray    : int
5 | $epais  : int      = 1
6 | $couleur : string = 'vert'
```

Comment lire cette signature ?

Tout d'abord il y a le *nom* de la fonction : `creer_cercle`.

Dans les parenthèses qui suivent, il y a les *arguments* de la fonction, c'est-à-dire ce que prend la fonction en entrée, lors d'un appel à cette fonction.

Ceux qui sont entre les crochets sont des *arguments optionnels*, c'est-à-dire que si on ne les fournit pas lors d'un appel à la fonction `creer_cercle`, ils prendront une valeur par défaut (définie en dessous). Il y a donc ici 2 arguments optionnels : l'épaisseur `$epais` et la couleur `$couleur`.

À la fin de la ligne, derrière le `->`, il y a le *type de retour* de la fonction, c'est-à-dire le genre de valeur que la fonction renvoie lorsqu'elle est appelée. En l'occurrence, il s'agit du type `cercle`, ce qui signifie que l'on fait l'appel suivant :

```
1 $x = creer_cercle(100, 100, 5, 1, 'bleu');
```

après l'exécution de cette instruction, la variable `$x` contient un cercle, ou plus exactement une certaine description d'un cercle, que l'on pourra utiliser ensuite dans d'autres calculs (par exemple pour transformer le cercle ou l'afficher). Ceci signifie que le cercle n'est pas

directement affiché à l'écran, mais qu'il est stocké dans `$x`.

S'il n'y a pas de type de retour, cela signifie que la fonction n'utilise pas l'instruction `return`, comme c'est souvent le cas avec les fonctions d'affichage.

En dessous de la première ligne de la signature de la fonction, il y a une description de chacun des arguments, avec leur type, ainsi que leur valeur par défaut s'il s'agit d'un argument optionnel.

Par exemple, l'appel suivant :

```
1 $x = cercle(100, 100, 5);
```

est équivalent à l'appel :

```
1 $x = cercle(100, 100, 5, 1, 'vert');
```

Il est possible de ne donner que certains des arguments optionnels, mais ceux-ci doivent être les premiers dans l'ordre des arguments. Par exemple :

```
1 $x = cercle(100, 100, 5, 2);
```

donne une valeur pour l'argument `$epaisseur`, mais laisse la valeur par défaut de `$couleur`.

A.2 Quelques fonctions et structures supplémentaires

Il existe de nombreuses fonctions en PHP, et vous pouvez y accéder à partir des URL fournies dans l'annexe C. En voici quelques-unes qui peuvent vous être utiles.

A.2.1 `print_r`

La fonction `print_r($mon_tableau)` permet d'afficher à l'écran le contenu du tableau passé en paramètre.

En effet :

```
1 $tableau = array('oui', 'non', 'peut-être');  
2 printline($tableau);
```

afficherait uniquement :

```
Array
```

ce qui n'est pas très informatif...

Alors que :

```
1 $tableau = array('oui', 'non', 'peut-être');
2 print_r($tableau);
```

affiche :

```
Array ( [0] => oui [1] => non [2] => peut-être )
```

A.2.2 rand

La fonction **rand** :

```
1 function : rand( $min , $max ) -> int
2 | $min : int
3 | $max : int
```

retourne un nombre entier entre les valeurs **\$min** et **\$max** incluses.

A.2.3 include

L'instruction **include**('mon-fichier.php'); permet d'insérer à l'endroit courant dans le fichier courant le contenu du fichier "mon-fichier.php".

Ceci est très pratique, notamment si l'on écrit des fonctions qui peuvent servir dans plusieurs fichiers.

A.2.4 break

L'instruction **break**; permet de terminer prématurément l'exécution d'une boucle. Elle ne peut exister qu'à l'intérieur d'une boucle.

A.2.5 isset

La fonction :

```
1 function : isset( $x ) -> bool
2 | $x : var
```

renvoie **true** si la variable **\$x** possède une valeur.

A.2.6 array_key_exists

La fonction :

```

1 function : array_key_exists( $clef , $tableau ) -> bool
2 | $clef      : string
3 | $tableau  : array

```

renvoie **true** si la clef `$clef` existe dans le tableau `$tableau`.

A.2.7 unset

La fonction :

```

1 function : unset( $x )
2 | $x : var

```

permet de supprimer la variable `$x`, de sorte que `isset($x)` retourne **false**.

A.3 Chaînes de caractères

Une chaîne de caractères est une suite de symboles qui sera interprétée par le programme comme du texte et non comme des commandes. Par exemple :

```

1 $texte = 'println("Bonjour !");';

```

ne fera que donner la valeur `'println("Bonjour !");'` à la variable `$texte` et n'écrira pas à l'écran "Bonjour!". Si par la suite on fait `println($texte);`, on affichera à l'écran :

```
println("Bonjour !");
```

A.3.1 Comparaison de chaînes de caractères

On peut tester si deux chaînes de caractères contiennent la même valeur avec l'opérateur `"=="`.

A.3.2 Guillemets simples et doubles, échappement de guillemet

Les guillemets doubles sont identiques aux guillemets simples à la différence qu'ils permettent de transformer les variables (et uniquement les variables simples, pas les indices dans les tableaux comme `$_GET['prenom']`) en leur valeur. Par exemple :

```

1 println("la valeur de x est $x.");

```

est équivalent à :

```
1 printline('la valeur de x est ' . $x . '.');
```

ce qui peut être très pratique dans la majorité des cas, mais nécessite parfois de faire attention à quelques pièges.

Pour afficher une apostrophe dans des guillemets simples, on la précède du symbole "\". On appelle cela de la désécialisation (car l'apostrophe a un rôle spécial dans une chaîne entre guillemets simples, rôle qu'on lui ôte ainsi) :

```
1 'Quelqu\'un a dit un jour : "À l\'origine de toute erreur attribuée à l
  \'ordinateur, vous trouverez au moins deux erreurs humaines. Dont
  celle consistant à attribuer l\'erreur à l\'ordinateur"'
```

Idem dans une chaîne de caractères entre guillemets doubles :

```
1 "Dave Small a dit : \"Un langage de programmation est une convention
  pour donner des ordres à un ordinateur. Ce n'est pas censé être
  obscur, bizarre et plein de pièges subtils. Ca, ce sont les
  caractéristiques de la magie.\""
```

Il n'y a pas besoin de désécialiser les guillemets simples dans une chaîne de caractères entre guillemets doubles et vice et versa.

A.4 Variables globales

Bien qu'il soit recommandé d'utiliser le minimum de variables globales possibles et d'utiliser au maximum le passage de valeur en paramètre des fonctions, il reste possible d'utiliser des variables globales, c'est-à-dire qu'elles conserve leur valeur au travers des différentes fonctions (mais au contraire des sessions, elles perdent leur valeur à la prochaine exécution du fichier).

En fait, toute variable peut être utilisée de manière globale. Il faut pour cela déclarer cette variable comme globale dans toute fonction l'utilisant :

```
1 <?php
2 function test_global() {
3     global $x, $y;
4     $y = 10;
5     printline($x);
6 }
7
8 function main() {
9     global $x, $y;
10    $y = 3;
11    $x = 5;
12    test_global();
13    printline($y);
14 }
15 ?>
```

Ce programme affichera bien 5 et 10, alors que sans le mot clef **global**, PHP aurait d'abord affiché une erreur :

```
Notice: Undefined variable: x in test.php on line 6
```

pour signaler que la variable **\$x** n'est pas définie, puis il affiche la valeur initiale de **\$y**, soit 3.

Plus que toute autre, une variable globale doit avoir un nom clair.

Il existe des variables particulières, dites *super globales*, comme les tableaux du type **\$_SESSION[]**, qui restent globaux par défaut, sans avoir besoin de préciser **global**.

La non-globalité est le comportement par défaut depuis la version 4.2.0 de PHP. Cependant, ceci peut varier selon la configuration et la version des serveurs Apache sur lequel est exécuté le programme PHP.

A.5 Gestion des erreurs

L'erreur la plus courante que vous verrez sera probablement :

```
Parse error: parse error, expecting ',' or ';' in
c:\mon_repertoire\monfichier.php on line 6
```

Petite explication :

```
Parse error:
```

signifie que PHP a détecté une erreur au moment de la lecture de votre fichier,

```
expecting ',' or ';' 
```

signifie qu'il manque une virgule ou un point-virgule,

```
in c:\mon_repertoire\monfichier.php
```

dans le fichier "mon_fichier.php",

```
Parse error: parse error, expecting ',' or ';' in  
c:\mon_repertoire\monfichier.php on line 6
```

au moment où PHP a atteint la ligne 6. Cela signifie que l'erreur se trouve sur la dernière instruction précédant la ligne 6. Il suffit probablement d'ajouter un point-virgule à la fin de la précédente instruction pour résoudre l'erreur.

Annexe B

Indentation en PHP

L'indentation est le fait de décaler les lignes de plusieurs espaces vers la droite pour rendre le programme plus lisible. On parle aussi de tabuler une ligne, c'est-à-dire le fait de la décaler vers la droite d'une tabulation, un appui sur la touche "tabulation".

Imaginez un roman où tous les mots seraient collés les uns aux autres. Cela donnerait la même difficulté de lecture qu'un programme non indenté.

Voici un exemple de code non indenté :

```
1 if($x==' a' ) {
2 $sortie = ' b' ;
3 if($y==' c' ) {
4 print (' d' );
5 } else {
6 $z = ' e' ;
7 }
8 } else {
9 print (' f' );
10 $sortie = ' g' ;
11 }
```

Ce qui est très peu lisible, et encore c'est un petit programme.

Voici le code précédent indenté :

```
1 if($x==' a' ) {
2     $sortie = ' b' ;
3     if($y==' c' ) {
4         print (' d' );
5     } else {
6         $z = ' e' ;
7     }
8 } else {
9     print (' f' );
10    $sortie = ' g' ;
11 }
```

Le sens du programme apparait tout de suite beaucoup plus nettement. Moins d'efforts sont nécessaires pour le comprendre.

Dans de nombreux éditeurs de textes, tel Notepad++, il est possible de sélectionner plusieurs lignes et d'appuyer sur (Shift-) "tabulation" pour les décaler d'une tabulation vers la droite (gauche).

Il ne faut pas non plus hésiter à sauter une ligne pour rendre la lecture du programme plus facile. Et surtout : mettre des commentaires tout au long du programme pour expliquer ce qu'il fait !

Annexe C

Ressources et aide en ligne

Il existe de très nombreuses ressources sur PHP, HTML, CSS, MySQL. Une simple recherche Google vous donnera dans les premiers résultats des pointeurs souvent pertinents.

Si vous connaissez le nom d'une fonction PHP mais que vous ne savez pas comment l'utiliser, le mieux est de faire une recherche Google du genre "php strpos" si vous cherchez de l'aide sur la fonction "strpos". Vous aurez tout ce dont vous avez besoin dès les premiers résultats.

Apprenez à être autonome ! Apprenez à utiliser par vous-même les moteurs de recherche, ce sont de puissants outils, et pas uniquement pour la programmation, mais dans tout domaine ¹.

Le site officiel français du PHP est :

<http://fr2.php.net/manual/fr/>

Il est parfaitement complet, en français et didactique. Veuillez vous y référer pour toute information complémentaire sur PHP.

Voici quelques tutoriels en français de PHP en ligne :

- **Commentcamarche** : <http://www.commentcamarche.net/contents/php/phpintro.php3>
- **Site du Zéro** : <http://www.siteduzero.com/tutoriel-3-14668-un-site-dynamique-avec-php.html>
- **Manuels PHP** : <http://www.manuelphp.com/manuels>

1. Attention toutefois à bien vérifier vos sources et à les croiser si besoin. Google et Wikipedia sont loin d'être infaillibles !

Annexe D

Utiliser Notepad++, PHP et MySQL chez vous

Cette annexe a pour but de vous permettre d'utiliser chez vous les outils du cours.

D.1 Éditeur de texte : Notepad++

Tout éditeur de texte (y compris le Bloc Notes de Windows, bien qu'il soit peu pratique) peut être utilisé pour écrire du HTML ou du PHP.

Notepad++ est un éditeur de texte plus pratique et possédant de nombreuses fonctionnalités. Il fournit entre autres la coloration syntaxique (les mots clefs sont en couleurs).

Vous pouvez le télécharger à :

<http://sourceforge.net/projects/notepad-plus/files/notepad%2B%2B%20releases%20binary/npp%205.4.5%20bin/npp.5.4.5.Installer.exe/download>

D.2 Navigateur : Firefox

Il vous faut un navigateur pour obtenir un rendu visuel du code HTML et pour faire l'interface avec le serveur Apache. Tout bon navigateur fera l'affaire (par exemple Internet explorer ou Firefox, avec une préférence pour ce dernier pour ses fonctionnalités et la qualité de son rendu).

D.3 Serveur Apache/Base de données : EasyPHP

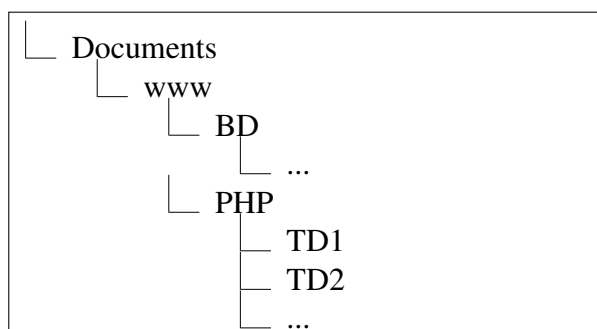
Pour le serveur Apache, qui sert à transformer les pages PHP en fichiers HTML et faire le lien avec le navigateur, ainsi que pour la liaison avec une base de données, il est recommandé d'utiliser EasyPHP, qui intègre tous ces outils.

Vous pouvez le télécharger à :
<http://www.easyphp.org/>

XAMPP est un logiciel similaire à EasyPHP qui peut être installé pour Windows, Linux ou Macintosh :
<http://www.apachefriends.org/fr/xampp.html>

D.3.1 Configuration d'EasyPHP

Pour utiliser correctement EasyPHP comme dans l'environnement de travail de cours, vous devez d'abord créer un répertoire de travail, disons "www", quelque part sur votre disque dur, peu importe, disons dans "Documents". Vous pouvez alors y créer la même arborescence des répertoires qu'en cours :



Lancez ensuite EasyPHP. Tout en bas à droite de Windows, à côté de l'horloge, vous devez avoir un petit "e" pour EasyPHP. Cliquez dessus avec le bouton *droit* de la souris. Un menu apparaît, choisissez "Administration". Ceci doit ouvrir une fenêtre dans votre navigateur Web. Dans la rubrique "Apache", sous-rubrique "Alias", cliquez sur "ajouter".

Dans le champ 2, écrivez "www" (sans les guillemets).

Dans le champ 3, entrez le chemin d'accès complet au répertoire "www" que vous avez créé, par exemple C : \Users\orseau\Documents\www.

Cliquez sur OK.

Maintenant, tant que EasyPHP est lancé, vous pouvez utiliser votre ordinateur de la même manière qu'en cours.

D.4 Paquetage AgroSIXPack

Vous pouvez récupérer le paquetage AgroSIXPack sur le site Web du cours. Il faut l'installer dans le répertoire que vous avez ajouté comme alias à Apache (cf. D.3.1).

Vos fichiers PHP doivent nécessairement se trouver dans un sous-répertoire du répertoire contenant le fichier AgroSIXPack.php.

Index

PHP

Affichage

`afficher_champ_paragraphe`, 49, 50
`afficher_champ_password`, 51
`afficher_champ_texte`, 49–51, 53, 62
`afficher_checkbox_item`, 49, 51
`afficher_checkbox`, 49, 51
`afficher_hr`, 46
`afficher_image`, 47
`afficher_lien_externes`, 47
`afficher_lien_interne`, 47, 56, 77
`afficher_liste_item`, 46, 48, 62
`afficher_radio_item`, 49, 52
`afficher_select_item`, 49, 52
`afficher_table_donnee`, 47–49
`afficher_table_entete`, 47
`afficher_table_ligne`, 46–48
`afficher_titre`, 46
`ajouter_champ_cache`, 52
`bold`, 46
`debut_formulaire`, 49, 50, 53, 62
`debut_groupe_checkbox`, 49, 51
`debut_groupe_radio`, 49, 52
`debut_liste`, 46, 48, 62
`debut_select`, 49, 52
`debut_table_ligne`, 47, 48
`debut_table`, 46, 48, 80, 81
`debut`, 56
`emph`, 46

`fin_formulaire`, 49, 50, 53, 62
`fin_groupe_checkbox`, 49, 51
`fin_groupe_radio`, 49, 52
`fin_liste`, 46, 48, 62
`fin_select`, 49, 52
`fin_table_ligne`, 47, 49
`fin_table`, 46, 49
`fin`, 57
`newline`, 12, 45
`print_r`, 73, 74, 87, 88
`printline`, 9, 10, 12–19, 21, 23–26, 28, 31–37, 39–43, 45, 49, 54, 56, 57, 60, 65, 68, 69, 71, 77, 87, 89–91
`print`, 10, 27, 28, 35–37, 39, 45, 93

Général

`GETcheckbox`, 54, 68
`GETref`, 50, 53–55, 63, 77
`agro_main`, 57
`and`, 19
`array_key_exists`, 55, 56, 89
`array`, 40–43, 46–48, 60, 72–74, 78, 80–82, 87–89
`break`, 88
`count`, 41, 72
`do`, 37, 38
`elseif`, 17
`else`, 16–18, 53, 54, 56, 93
`false`, 11, 16, 18, 19, 36, 49, 51–54, 60, 69, 78, 79, 89
`foreach`, 41–43
`for`, 38, 39, 41
`function`, 9, 10, 13, 20, 21, 23, 24, 28, 30–33, 37, 45–62, 67, 71–74, 76–82, 86, 88, 89, 91

- global**, 91
 - if**, 16–19, 35, 37, 39, 53, 54, 56, 60, 63, 93
 - include**, 57, 88
 - isset**, 88, 89
 - main**, 9, 10, 13, 21–24, 28, 32, 33, 37, 44, 45, 49, 53, 54, 56, 57, 62, 65–68, 71, 76–82, 91
 - or**, 19
 - rand**, 72, 88
 - redirection**, 55, 77
 - return**, 20–24, 28, 33, 87
 - true**, 11, 16, 18, 19, 36, 49, 51, 53, 54, 60, 69, 78, 79, 88, 89
 - unset**, 89
 - while**, 35–39, 41, 53, 60, 62
 - xor**, 19
- MySQL
- mysql_close**, 58, 61, 63
 - mysql_connect**, 58, 63
 - mysql_fetch_assoc**, 60, 62
 - mysql_query_affiche**, 59, 61, 62
 - mysql_query_debug**, 59, 60
 - mysql_query**, 59, 60
 - mysql_real_escape_string**, 61, 62
 - mysql_select_db**, 58, 59, 63
- Opérateurs
- Opérateurs d’affectation, 14
 - Opérateurs de comparaison, 18
 - Opérateurs logiques, 19