

R1.04 - Systèmes - TP 9

Scripting PHP

Introduction

Qu'est-ce que PHP ?

PHP est un langage initialement développé pour le Web et il signifiait, au début, **P**ersonal **H**ome **P**age¹.

Il a donc été estampillé, à tort, comme un langage exclusivement Web.

Mais **PHP** est avant tout un langage de scripting comme un autre, comme **Javascript**, **Python**, et bien d'autres, et notamment **Bash** que nous avons vu récemment.

Effectivement, **PHP** est très adapté au codage pour le Web. Par exemple, il est capable de traiter facilement les **<FORM>** soumis depuis une page Web ou encore les **COOKIES** d'un site. Mais ça n'implique pas qu'on doive le rendre exclusif à un domaine, celui du Web.

Que vient faire PHP en Système ?

Le scripting en Système a, depuis l'origine, été fait à l'aide d'outils datant d'une époque où il n'existait que le Shell et quelques autres commandes ou filtres dédiés à l'administration Système.

Scripter en langage Shell, tel que **bash**, est aussi l'assurance qu'un script pourra s'exécuter sur n'importe quel système Unix, même minimaliste, puisque le Shell est lui-même un outil nécessaire pour administrer le système.

Cependant, comme nous avons pu le voir dans le TP précédent, **bash** est assez austère, dans sa syntaxe notamment.

Nous avons donc fait le choix de vous apprendre les bases du langage **bash** pour les besoins quotidiens légers (enchaînements de quelques commandes avec un ou deux tests, quelques variables, etc.). C'est chose faite depuis le TP 8.

Nous avons aussi décidé d'aller plus loin dans le scripting en utilisant un autre langage, plus moderne et plus simple, très souple et très puissant. Le choix s'est porté sur PHP car

¹ Il signifie maintenant **PHP Hyper Processor**. Vous aurez noté le caractère récursif de cette appellation... c'est de l'humour d'informaticien, comprenez qui peut 😊

c'est aussi un langage très courant dans le monde professionnel et que vous étudierez en cours de Web Côté Serveur à l'IUT. Ainsi vous capitalisez sur l'avenir.

Cependant, nous n'aborderons que les bases du PHP, des bases qui sont largement suffisantes pour les besoins du quotidien. Vous verrez PHP plus en profondeur dans le cours dédié à PHP, cité précédemment.

L'inconvénient est que PHP ne figure pas parmi les outils que vous êtes sûr de trouver sur tout système Linux. Il faudra alors passer par une installation de l'interpréteur **PHP** (en mode **CLI**²), une contrainte à prendre en considération quand vous aurez à choisir le type de script que vous voulez écrire. Mais c'est une contrainte acceptable³ la plupart du temps.

Premiers pas

Vous avez le droit d'utiliser VSC pour écrire vos scripts. Vous les lancerez à la main dans un Terminal (dans VSC ou dans le Terminal Linux)

Bonjour qui ?

Devinez à qui on va encore dire bonjour...

Voici votre 1^{er} programme en PHP, tapez ceci dans un fichier **hello.php** :

```
<?php
    echo "Hello World!\n"
?>
```

Puis, dans le Terminal (ouvert dans le bon dossier), tapez ceci :

```
php hello.php
```

Vous devez avoir le fameux affichage auquel vous êtes désormais habitué.

Shebang

On a eu l'occasion de voir le shebang pour **bash**, généralement c'est un : **#!/bin/bash**

² Command Line Interface (ligne de commande)

³ Même pour l'administrateur à qui vous ferez la demande !

Avec le shebang, on indique au Shell où se trouve l'interpréteur de script qui sera capable d'exécuter le reste du script. Il lance cet interpréteur, puis lui donne à manger le script, c'est-à-dire tout le reste, à partir de la ligne 2 du fichier script.

Nous allons faire de même pour ce script PHP. L'interpréteur **PHP** s'appelle... **php** bien entendu, nous venons de le lancer dans l'exercice précédent !

Il nous faut d'abord localiser où se trouve cet interpréteur. C'est aussi une commande finalement. Pour cela, tapez ceci dans votre Terminal :

```
which php
```

Il ne vous reste plus qu'à ajouter le chemin vers l'interpréteur en 1^{ère} ligne (avant la partie **<?php ...**) de votre script **hello.php** :

```
#!/placez_ici_le_resultat_du_which_php
```

Maintenant vous allez pouvoir lancer **hello.php** directement, comme vous feriez avec un script **bash** :

```
chmod +x hello.php  
./hello.php
```

Et voilà !

Notez que le **chmod +x** n'est à faire qu'une seule fois par script.

<?php ... ?>

Héritage de son usage initial pour le Web, un script PHP doit obligatoirement être placé entre un tag ouvrant **<?php** et un tag fermant **?>**

Tout ce qui est placé en dehors de ces tags ne sera pas interprété comme du code PHP mais s'affichera simplement à l'écran. Sans entrer dans le détail, ce comportement a de l'intérêt pour créer des pages Web mais n'en a aucun dans notre cas.

Retenez simplement que la ligne sous le shebang doit commencer par un **<?php** et que la dernière doit terminer par un **?>** et qu'aucun de ces tags ne doit apparaître ailleurs dans votre code. On place généralement ces deux tags seuls, sur leurs lignes respectives, et l'intérieur, qui est donc votre script, est indenté.

Le script **hello.php** est construit de cette manière.

Pour expérimenter tout ce qui va suivre sans créer un script à chaque fois, vous allez lancer **php** en mode *Shell Interactif* et un peu comme dans un **bash**, vous pourrez taper directement les instructions **PHP** qui vont suivre et observer immédiatement le résultat.

Pour lancer un **php** en *Shell Interactif*: **php -a**

Pour le quitter une fois tout terminé tapez la combinaison de touches **CTRL+D** ou **exit**

Si vous êtes bloqué dans une commande, tapez **CTRL+C** et relancez un **php -a**

Syntaxe

Comme de nombreux langages, PHP hérite fortement du langage C concernant sa syntaxe de base mais sans les côtés complexes et contraignants du C.

Nous n'allons pas étudier PHP dans toutes ses finesses, ce sera fait plus en détail en Programmation Web Côté Serveur. Nous allons nous limiter aux bases du langage et aux éléments clés pour écrire des scripts pour l'administration Système.

Comme en C, les instructions sont terminées par des **;** (points-virgules).

Types de données

Voici quelques types de données (de variables) que nous allons utiliser :

- Entiers : **123**
- Flottants : **123.456**
- Chaînes de caractères : **"ABC"** ou aussi **'ABC'**
- Booléens : **true** et **false**
- Tableaux : voir plus loin

Variables

Comme avec beaucoup de langages de scripting, on l'a vu notamment avec **bash**, les variables ne nécessitent pas d'être déclarées avant de pouvoir les utiliser.

Une variable porte un nom qui commence par une lettre (majuscule ou minuscule) ou un **_** (underscore), suivi de lettres (majuscule ou minuscule), de chiffres ou de **_**

Une variable est toujours préfixée d'un **\$**, que ce soit pour la lire ou pour l'affecter.

Affectation

```
$nom = "Sarah Connor";  
$salaire = 2500;
```

le nom de la variable à affecter doit être préfixé d'un **\$**

La variable est créée uniquement au moment de son affectation.

Lecture

L'utilisation du contenu d'une variable se fait simplement en utilisant ***\$variable*** dans l'expression. Exemples :

```
echo $nom;  
echo $salaire * 12;  
$annee = $salaire * 12;  
echo $annee;
```

Utiliser une variable inconnue dans une expression ne provoque pas d'erreur et ne crée pas la variable non plus. Seule l'affectation d'une variable (encore) inconnue, créera la variable.

En cas d'utilisation d'une variable inconnue dans une expression, une valeur par défaut est utilisée à la place de la variable inconnue. Cette valeur dépend du contexte de l'expression où elle est utilisée.

Exemple 1 :

```
echo $variable_inconnue * 12;
```

Affiche **0**, car en apparaissant dans une expression arithmétique, la ***\$variable_inconnue*** est donc remplacée par un **0** dans cette expression.

Exemple 2 :

```
echo "ABC" . $variable_inconnue . "DEF";
```

Affiche **ABCDEF**, car en apparaissant dans une expression de chaîne, la ***\$variable_inconnue*** est donc remplacée par un **""** (chaîne vide) dans cette expression.

NB : l'opérateur **.** (point) est une concaténation de chaînes.

Opérateurs

Nous allons passer les principaux en revue sans nous attarder sur chacun car vous les connaissez déjà dans le C.

Arithmétiques

+ **-** ***** **/** **%** (modulo euclidien) ****** (exponentiation)

Exemples :

```
$val1 = 5;
$val2 = 8;
echo 7 * 12 + $val1 / $val2;
```

Pré et Post-(in|dé)crémentation

++ **--**

Exemples :

```
$val=10;
echo ++$val;
echo $val;
echo $val--;
echo $val;
```

Opérateur logiques

&& **||** **!**

Exemples : voir le paragraphe sur les comparaisons.

Opérateur de chaîne

. (point, c'est une concaténation)

Exemples :

```
echo $nom;
echo "Mme " . $nom;
```

Opérateur avec réaffectation

+= -= *= /= %= .=

Exemples :

```
echo $val;
$val += 12;
echo $val;
$val /= 5;
echo $val;
$ville = "Lannion";
$ville .= " (France)";
echo $ville;
```

Priorité des opérateurs

La priorité des opérateurs est la même qu'en C et on peut aussi utiliser des **()**.

Structures de Contrôle

Alternatives

(Exemple de syntaxe, ne pas tester dans l'Interactive Shell **PHP**)

```
if (condition) {
    ...
} else if (condition2) {
    ...
} else {
    ...
}
```

Le **if** est obligatoire mais le ou les **else if** et le **else** sont facultatifs.

Un **else if** peut aussi s'écrire **elseif** et fonctionner de la même façon.

Comme en C, les `{}` ne sont obligatoires que s'il y a plus d'une instruction à exécuter mais elles sont très fortement conseillées pour une question de propreté et de clarté du code.

La condition fait entrer en jeu les opérateurs de comparaison et les opérateurs logiques. Voir le paragraphe dédié plus loin.

Boucles **for**

```
for ($boucle = 1; $boucle < 10; $boucle++) {  
    echo $boucle . "\n";  
}
```

Très semblable au C, cette structure doit se comprendre sans effort.

Boucles **while** et **do... while**

(Exemples de syntaxe, ne pas tester dans le Shell Interactif **PHP**)

```
while (condition) {  
    ...  
}
```

```
do {  
    ...  
} while (condition);
```

Très semblables au C, ces structures doivent se comprendre sans effort.

Conditions et comparaisons

Les alternatives et les boucles font apparaître des **conditions**.

Les conditions sont des expressions booléennes qui s'écrivent avec des opérateurs de comparaison et des opérateurs logiques.

Opérateurs de comparaison

`==` `!=` `<=` `>=` `<` `>`

Ces opérateurs, qui sont les mêmes que ceux qu'on trouve en C pour les valeurs numériques, peuvent aussi s'utiliser sur des chaînes de caractères en PHP !

Sur une chaîne de caractères, un `==` est la même chose en PHP qu'un `strcmp()` en C, ce qui est plutôt sympathique. Dans ce cas un `<=` ou un `>=`, qui permettent habituellement de classer des valeurs numériques, sauront classer des chaînes de caractères entre elles, par ordre alphabétique pour établir laquelle et la plus petite par exemple.

Mixage des types

Les expressions font entrer en jeu des variables ou des expressions qui peuvent être :

- Numériques
- Chaînes de caractères
- Booléennes

On vient de voir que PHP est plus souple que le C en permettant d'utiliser les mêmes opérateurs de comparaison qu'il s'agisse de valeurs numériques ou de chaînes de caractères. Mais il est encore plus souple dans sa façon d'interpréter les types de données, car il autorise le mixage des types.

Par exemple, on peut comparer des chaînes de caractères avec des valeurs numériques, sans avoir besoin de transformer les types de données. Ainsi :

```
if ((4 * 3) == "12") {  
    echo "OK";  
}
```

compare un résultat d'une expression numérique (`4 * 3`) avec une chaîne de caractères, sans que ça lui pose de problèmes.

Il transforme la chaîne `"12"` en une valeur numérique (`12`), puis il fait le test d'égalité avec le résultat de `4 * 3`, c'est-à-dire la valeur numérique `12`, aboutissant à évaluer à **true**.

A noter que ceci :

```
if ((4 * 3) == "00012ABC") {  
    echo "OK";  
}
```

fonctionne aussi et est aussi évalué à **true**. La raison réside dans la façon dont PHP convertit une chaîne en valeur numérique quand il a besoin de le faire, comme ici, afin de comparer avec une autre valeur numérique.

PHP extrait de la chaîne tout ce qui forme une valeur numérique, en partant de la gauche, jusqu'à rencontrer une impossibilité d'aller plus loin. Ici c'est la rencontre du caractère **A** qui arrête la conversion, pour ne garder que ce qu'il a réussi à extraire : **12** (les **0** initiaux, ne servant à rien, n'ont aucun impact).

Attention, il y a quelques caractères autres que des chiffres qui sont acceptables dans une chaîne :

```
echo 10 * "1e2";    // Affiche 1000
echo 2 + "+1e-2";   // Affiche 2.01
echo 5 / ".1";      // Affiche 50
```

car **1e2** est une écriture de **100** en notation scientifique, **+1e-2** est une écriture de **0.01** et **.1** est une écriture de **0.1**

Comparaisons strictes

Tout ceci est assez séduisant mais provoque un effet de bord qu'il est important de bien comprendre.

Comme PHP est souple et convertit les types pour que les comparaisons se fassent d'une façon ou d'une autre, on aboutit alors à ce que les valeurs :

0 0.0 false ""

sont toutes considérées comme égales par **PHP**. Ainsi :

```
if ("" == false) {
    echo "OK";
}
```

est vrai (c'est à dire qu'une chaîne vide vaut bien **false**), comme ceci est vrai aussi :

```
if (0 == "") {
    echo "OK";
}
```

Si si ! Il suffit de réfléchir à la façon dont PHP convertit **""** en une valeur numérique. On a vu qu'il s'arrête quand il rencontre un caractère qu'il ne sait pas convertir. Ici il s'arrête tout de suite, puisque la chaîne est vide et il utilise ce qu'il a pu convertir jusqu'à ce point. Mais il part bien d'une valeur de départ, qui est zéro, donc **""** vaut bien **0** !

Pourquoi cela pose-t'il problème, parfois ? Simplement parce qu'il peut arriver de souhaiter qu'une valeur numérique soit considérée comme quelque chose de différent d'une chaîne de caractère, même si son contenu "ressemble" à une valeur numérique.

Il nous apparaît évident que si **A = B** et que **B = C**, alors **A = C**. Mais c'est sans compter sur les comparaisons souples de **PHP**, qui nous mènent à ce paradoxe (ne rien saisir) :

```
| "0" == 0
```

est vrai,

```
| "" == 0
```

est vrai, on peut donc en déduire que, logiquement,

```
| "0" == ""
```

doit être vrai, non ?! Pourtant c'est évidemment faux, les deux chaînes sont bien sûr différentes.

PHP a donc prévu une solution pour résoudre ce problème, et éviter de tomber dans ce paradoxe, si besoin. La solution s'appelle la *comparaison stricte*.

Une comparaison stricte compare non seulement les valeurs mais aussi les types. Deux valeurs (souvent deux variables) ne peuvent être égales que si elles sont de même type, sinon la comparaison évalue à faux. Ces comparaisons ne concernent que l'égalité ou la différence.

L'opérateur d'égalité stricte est : **===**

L'opérateur de différence stricte est : **!==**

Ainsi (ne rien saisir) :

```
| "10" === 10  
| 0.0 === 0  
| 0 === false
```

évaluent tous à faux.

Tableaux

Les tableaux sont des composants essentiels de PHP. Ils participent grandement à la flexibilité et la puissance de PHP.

Il y a plus de 80 fonctions dédiées aux tableaux dans le langage **PHP**.

Ces tableaux seront souvent les éléments centraux des scripts que vous allez coder ici et plus tard, en programmation Web.

Vous connaissez les tableaux en C :

```
int vals[10];
```

En C, un tableau contient un nombre de cellules fixé dès le départ (ci-dessus : **10**) et chaque cellule est d'un type défini, lui aussi, dès le départ (ci-dessus : **int**).

En C, les cellules d'un tableau sont accessibles par un indice numérique allant de **0** à **N-1**, où **N** est la taille du tableau (ci-dessus : **0** à **9**).

En PHP, un tableau est dit *associatif*, car chaque cellule associe une clé (numérique ou chaîne de caractères) à une valeur de n'importe quel type.

En PHP, un tableau est élastique et grossit quand on lui ajoute des cellules ou rétrécit quand on en retire.

En PHP, les cellules d'un tableau peuvent être de types différents.

Déclaration

A l'inverse des autres types de données qui ne nécessitent pas de déclarer une variable avant de l'utiliser, certaines actions nécessitent qu'un tableau soit défini avant de pouvoir être utilisé. Le plus simple est donc de toujours définir une variable de type tableau avant de l'utiliser :

```
$tablo = array();  
$tablo2 = [];
```

sont deux syntaxes pour définir un tableau vide. La seconde est la plus moderne et on retiendra celle-là désormais.

Affectation

En même temps qu'on déclare un tableau, on peut lui affecter un contenu de la façon suivante :

```
$depts = [  
    22 => "Côtes d'Armor",  
    29 => "Finistère",  
    35 => "Ille-et-Vilaine",  
    56 => "Morbihan",  
];
```

L'association de la clé et de sa valeur se fait par une flèche =>

Astuce

PHP dispose d'une fonction très pratique pour afficher le contenu d'une variable.

Pour une variable simple, comme une chaîne de caractères ou une valeur numérique, ça n'apporte rien de plus qu'un simple **echo**, mais avec les tableaux c'est bien plus pratique car ça affiche tous les éléments du tableau (valeurs et clés associées), et cela en profondeur : si un tableau contient des tableaux qui contiennent des tableaux etc. Alors qu'un **echo** sur un tableau affichera simplement : **Array**.

La fonction s'appelle **print_r()**, et voici sa syntaxe :

```
print_r($depts);
```

On observe que, même si les clés sont numériques, ce ne sont pas des indices, il peut y avoir des trous. La clé 22 ne signifie pas la 22^{ème} (ou 23^{ème}) cellule du tableau, comme ce serait le cas en C.

Les clés d'un tableau sont uniques et les valeurs quelconques. Autre exemple :

```
$etu = [  
    'f1' => [  
        "Pipo",  
        "Lulu",  
        "Jako",  
    ],  
    'f2' => [  
        "Pipo",  
        "Lulu",  
        "Jako",  
    ],  
];
```

```
        "Riri",  
        "Fifi",  
        "Loulou",  
    ],  
];  
  
print_r($etu);
```

Dans cet exemple, chaque cellule du tableau **\$etu** est aussi un tableau (de chaînes).

Observez qu'à la déclaration-affectation des sous-tableaux, on n'a pas indiqué de clés pour les noms. Le **print_r()** en affiche pourtant et elles sont numériques. Si on ne précise pas de clé, le tableau associatif est en tout point similaire à un tableau indicé, comme en C.

Ajout/Remplacement

Ajouter une cellule à un tableau associatif se fait simplement comme l'affectation d'une variable.

Si la cellule existe déjà (si la clé existe déjà), la valeur de la cellule est remplacée :

```
$depts[22] = "Aodou An Arvor";  
  
print_r($depts);
```

Si la cellule n'existe pas encore (si la clé n'existe pas), la valeur de la cellule est créée, le tableau est agrandi :

```
$depts[44] = "Loire Atlantique";  
  
print_r($depts);
```

Parcours

Il a été dit précédemment que les tableaux sont des éléments de PHP.

Les tableaux viennent avec une structure de contrôle qui leur est dédiée : **foreach**

Voici sa syntaxe :

```
foreach ($depts as $num => $nom) {  
    echo "Le département " . $num . " s'appelle " . $nom . "\n";  
}
```

qu'on peut traduire par : *dans une boucle, pour chaque cellule du tableau contenu dans la variable **\$depts**, place sa valeur dans la variable **\$nom** ainsi que la clé associée à cette valeur dans la variable **\$num***. Évidemment **\$num** et **\$nom** peuvent être nommées avec les noms que vous voulez.

Vous noterez que les affichages se font par concaténations de chaînes et de variables.

Si la clé ne vous est d'aucune utilité, une forme raccourcie peut alors être :

```
foreach ($depts as $nom) {  
    echo $nom . "\n";  
}
```

En PHP, quand vous avez un tableau, vous avez, à coup sûr, besoin d'un **foreach()** !

Fin de ligne

Notez que, dans les deux exemples précédents, chaque affichage se fait avec un retour à la ligne automatique ajouté par `. "\n"`, comme on aurait besoin de le faire aussi dans un `printf("...\n")` du C.

Au contraire, le **echo** du **bash** fait un retour à la ligne tout seul. Si on souhaite ne pas avoir ce retour à la ligne par défaut, le **echo** du **bash** a besoin d'une option **-n**.

Chaînes : " vs '

En PHP, comme avec **bash**, les chaînes peuvent s'écrire entre " (guillemets) ou ' (apostrophes) avec les mêmes différences (liste non exhaustive) :

- Avec les " : utilisation possible des variables, et un `\n` devient un retour à la ligne.
- Avec les ' : le `$` reste le symbole du dollar, donc utilisation impossible des variables, et un `\` (backslash) reste un symbole `\`. Donc un `\n` est un `\` suivi d'un `n`.

Conclusion

Voilà, on n'ira pas plus loin dans l'étude du langage PHP. Nous avons suffisamment de matière pour les besoins de scripting en Système.

Sachez que PHP est un langage qui est accompagné d'une librairie de fonction extrêmement complète : gestion et accès aux BDD, gestion de fichiers, gestion de fichiers compressés, Crypto, Maths, Traitement d'images, évidemment tout ce qui touche à Internet (Web, Mail, FTP, etc.), pour n'en citer qu'une partie.

Vous en verrez plus en Web Côté Serveur.

Passons à la mise en pratique de tout ceci.

Ateliers

Vous allez maintenant écrire des scripts, sauf indication contraire, ne travaillez plus dans le Shell Interactif **PHP**.

Nommez vos scripts du nom de la question.

Les fichiers de travail sont ceux qu'on a déjà utilisés dans les TP précédents.

Pour la syntaxe du langage PHP, vous avez tout ce qu'il vous faut dans la 1^{ère} partie de ce TP.

Pour la documentation des fonctions de PHP (elles vous seront données dans le sujet), regardez la documentation officielle : <https://www.php.net/manual/fr/funcref.php> (il y a un champ de recherche en haut de page pour un accès direct à une fonction)

On va utiliser PHP **UNIQUEMENT** pour réaliser des traitements qu'on ne peut pas faire facilement avec des filtres.

Si vous devez compter des lignes d'un fichier, en extraire des champs, transformer les : (deux-points) en ; (points-virgules), ou encore récupérer les 100 dernières lignes, etc., ne vous lancez pas dans un script en PHP, il y a des filtres qui savent faire cela très bien et très rapidement.

Si vous avez besoin de calculer la moyenne de 3 champs de chaque ligne paire, et d'en faire la somme totale uniquement des moyennes négatives, ça ressemble à quelque chose de suffisamment compliqué pour qu'une série de filtre n'en vienne pas à bout. PHP sera alors un bon candidat pour vous venir en aide, et même cette tâche qui semble complexe pourra se réaliser assez simplement.

Nous verrons à la fin de ce TP comment on peut écrire des scripts PHP qui pourront compléter les filtres et s'insérer en toute transparence dans des successions de tubes, comme les filtres natifs de Linux.

- Q.1 Ecrivez et testez un script qui fait la somme (**5050**) des entiers de **1** à **100** (par une boucle) et affiche le résultat.
- Q.2 Vous allez écrire et tester un script qui affiche les (**6**) lignes du fichier **murphy** qui font moins de **60** caractères de long. On va faire ce 1^{er} script en conduite accompagnée...

Étapes :

- **Étape 1** - Charger les lignes du fichier dans un tableau (1 ligne = 1 cellule du tableau). Pour ce faire, la fonction **file()** va vous être très utile car elle fait exactement cela. Syntaxe (jetez aussi un œil sur le site de la documentation) :

```
$lines = file("nom_fichier");
```

Seul le 1^{er} paramètre est nécessaire. L'avantage est que vous n'avez pas besoin d'ouvrir le fichier, de le lire ligne par ligne, de stocker dans un tableau chaque ligne lue et de refermer le fichier. La fonction **file()** fait tout ça elle-même. PHP est souvent ainsi, il fait une grosse partie du travail pour vous !

La fonction retourne un tableau, qui est stocké dans votre variable **\$lines** (qu'il crée d'ailleurs en même temps).

Comme vous êtes en conduite accompagnée, l'accompagnateur dit : *il faut mettre le bon nom de fichier.*

- **Étape 2** - Parcourir chaque cellule du tableau et pour chaque ligne, calculer la longueur de la cellule courante (donc la ligne courante) et tester si la longueur est ≤ 60 . Si c'est le cas, on l'affiche, sinon on continue (donc ça boucle). Calculer la longueur d'une chaîne de caractères se fait ainsi :

```
$len = strlen(...);
```

L'accompagnateur dit : *on parcourt un tableau avec un **foreach(...)** et il dit aussi : il faut mettre la bonne chaîne dans **strlen()** et cette chaîne est simplement la cellule du tableau qui provient du **foreach(...)**.*

Avant de retourner dans un sommeil profond sur son siège passager, l'accompagnateur termine en disant : *on n'a pas besoin d'utiliser la clé de chaque ligne.*

Ce script que vous venez de créer est la base de tous les scripts que vous allez écrire ci-après. Si besoin, c'est le moment de vous manifester auprès de votre enseignant.e.

Fin de ligne (suite)

Si vous avez suivi à la lettre la façon de coder en PHP apprise jusqu'ici, vous avez certainement ajouté un . `"\n"` dans vos affichages des lignes de **murphy**, dans l'exercice ci-avant. Et vous avez rencontré un phénomène de double saut de ligne.

En fait, quand **file()** lit un fichier en plaçant chaque ligne dans un tableau, chacune de ces lignes contient aussi un `\n` final (lu depuis le fichier) qui est conservé. C'est pour cela que le **echo \$line** (ou tout autre nom que vous aurez utilisé dans votre **foreach()** des lignes de **murphy**) n'a pas besoin d'un ajout de `\n`.


Ca peut sembler pratique mais il est quand même conseillé de toujours retirer le `\n`, quitte à le rajouter si besoin pour l'affichage. Il vous sera plus souvent problématique de le conserver que de le supprimer. Voici le code conseillé dans ce cas :

```
foreach ($arr as $key => $val) {  
    $val = rtrim($val);  
    ...  
}
```

La commande **rtrim(...)** permet de supprimer tous les `\n` ou `\r` de la fin d'une chaîne de caractère. On réaffecte la variable par sa version éventuellement nettoyée.

Il est conseillé de toujours effectuer ce nettoyage par **rtrim()**. Notez que **trim()** fait la même chose mais des deux côtés : au début et à la fin d'une chaîne de caractères.

- Q.3 Sur la base du script précédent, écrivez et testez un script qui compte le nombre de mots sur chaque ligne et affiche le numéro de chaque ligne et le compte de mots.

 La fonction **explode()** permet de découper une chaîne de caractères sur un séparateur donnée et retourne un tableau dont chaque cellule est un des morceaux découpés. Jetez-y un œil sur le site de documentation, puis testez ceci dans un Shell Interactif **PHP (php -a)** :

```
$parts = explode(":", "James:Bond:MI6");  
print_r($parts);
```

Avec cette nouvelle fonction qui, comme vous le voyez, retourne un tableau, vous pouvez compléter le script pour compter le nombre de mots de chaque ligne de **murphy**. Par simplicité, on considère uniquement l'espace comme séparateur de mots.

 Les fonctions **count()** et **sizeof()** permettent de connaître la taille d'un tableau.

Q.4 Vous avez peut-être remarqué que le filtre **cut**, qui permet d'extraire des champs, le fait en conservant l'ordre qu'ils ont dans le fichier non pas celui qu'on a indiqué sur la ligne de commande : si on écrit **cut -f5,3,1** on obtient les champs dans l'ordre **1, 3** et **5**, ce qui est bien dommage, et on peut même considérer que c'est une fonctionnalité oubliée. Qu'à cela ne tienne, on va s'arranger avec PHP.

Affichez donc ces colonnes **5, 3** et **1** du fichier **prod**, en respectant cet ordre et en séparant les champs par un **;** (point-virgule).

Q.5 Le fichier **demog** contient des données démographiques de l'Europe par périodes de 5 ans. Voici son contenu par colonne :

- Année
- Population
- Naissances annuelles
- Décès annuels

Affichez les lignes complètes en ajoutant les colonnes supplémentaires suivantes, en gardant le même séparateur :

- Le solde naturel annuel : Naissances - Décès
- Le taux de natalité : Naissances / Population, exprimé en *pour mille*.
- Le taux de mortalité

Pour information, voici les valeurs complémentaires que vous devez trouver pour **1960** :


- **5088682**
- **20,08**
- **9,32**

 La fonction **round()** permet d'arrondir une valeur flottante (cf doc).

Q.6 Sur la base du fichier **demog**, affichez uniquement la somme totale des (**44746341**) naissances et des (**34384773**) décès des années divisibles par **10** (les années finissant par **0**).

Q.7 Reprenez le fichier de logs Web **access.log**, et affichez les adresses **IP** uniques et la somme des tailles des pages consultées en groupant par l'adresse **IP**.

Pour information, Il y a **484 IP** différentes et l'**IP 13.66.139.0** a consommé **32653** octets.

 Un tableau associatif, comme le sont les tableaux PHP, ne peut avoir que des clés uniques. Utilisez un tableau de résultat dont les clés sont les **IP**. Si une clé n'existe pas, essayer d'accéder à la cellule associée retournera simplement **0**. En fin de script, il vous

restera simplement à parcourir le tableau de résultat pour afficher les valeurs (garanties d'être uniques) au bon format attendu.

Q.8 En utilisant le script de la question précédente, affichez le résultat trié par taille décroissante, en utilisant un filtre pour Linux en complément du résultat produit par le script PHP. Identifiez l'**IP (62.210.207.209)** qui a consommé le plus de bande passante.

Q.9 Affichez les lignes du fichier **depts** en formatant chaque ligne ainsi :

Côtes d'Armor (22)
Finistère (35)

Attention au piège, voyez-vous le **Territoire de Belfort (90)** dans votre liste ?

Q.10 Même question que précédemment, mais en ne gardant que les **(96)** lignes uniques, car il y a des départements en double dans le fichier.

Bonus important

PHP & STDIN

Héritage Web

Rappelons-le, PHP vient du monde du Web.

Son premier usage a été de créer dynamiquement des pages HTML, c'est-à-dire construites à la volée, avec des parties statiques (qui sont les mêmes pour tout le monde, l'habillage de la page, les menus, le bandeau de tête etc.) et des parties variables telles que le contenu spécifique à l'utilisateur, par exemple : l'affichage du contenu du panier d'un client sur un site marchand. Cet affichage est dynamique car il est fonction de ce que le client y a ajouté au gré de sa navigation.

Sur un site Web, l'utilisateur est devant son navigateur Web. En reprenant l'exemple du client et du site marchand, l'utilisateur saisit des choses dans des champs de formulaire ou il clique sur des liens et remplit ainsi son panier. Ces interactions se font dans le navigateur, c'est-à-dire à des kilomètres de l'endroit où le script PHP va s'exécuter. En effet, dans le cas du Web, un script PHP s'exécute sur le serveur Web, pas sur l'ordinateur du client.

Comment le script obtient-il les données saisies par l'utilisateur ?

C'est le navigateur qui fait la passerelle entre l'utilisateur et le serveur Web. Quand l'utilisateur clique sur un bouton de formulaire ou sur un lien, le navigateur transfère les informations au serveur Web qui exécute le script PHP prévu pour le traitement, en lui passant en même temps les informations reçues du navigateur.

Si on vous explique tout ceci alors qu'on n'est pas dans un cours sur la programmation Web Côté Serveur, c'est justement pour que vous compreniez que l'usage qu'on fait de PHP en Système est un peu différent : ici, l'utilisateur peut être devant son écran et son clavier, devant le Terminal qui exécute le script.

Or, du fait de l'héritage Web du PHP et de la façon dont le Web fonctionne, comme on vient de le voir, rien n'a été prévu pour lire des choses au clavier puisqu'en principe c'est un navigateur Web qui sert de passerelle pour faire transiter les données saisies (dans des formulaire par exemple) jusqu'au script.

STDIN

Comme tout programme ou commande, **php** a aussi un **STDIN** et un **STDOUT**⁴.

Vous avez déjà expérimenté le **STDOUT** d'un script PHP avec **echo** et aussi **print_r()**. Il existe d'autres fonctions pour afficher que nous ne verrons pas ici.

Concernant **STDIN**, on vient de voir et d'expliquer pourquoi rien n'est prévu, en standard, pour l'utiliser, pour lire au clavier.

Est-ce que ça signifie que c'est impossible ? La réponse est non, on peut le faire mais au prix d'une petite gymnastique, un petit *tweak* !

Voici le code, très simple, d'une fonction **read()** qui est similaire en tout point à l'instruction **read** que vous utilisez en **bash** :

```
function read() {  
    static $stdin = null;  
  
    if ($stdin === null) {  
        $stdin = fopen('php://stdin', 'r');  
    }  
    return rtrim(fgets($stdin));  
}
```

Vous devez placer le code de cette fonction **read()** en début (ou fin) de vos scripts qui ont besoin de lire au clavier (**STDIN**).

On ne vous demande pas de comprendre comment fonctionne ce code, même si ça n'a rien de compliqué en soi. Vous pouvez simplement le copier-coller et l'utiliser en l'état.

NB : vous pouvez la renommer si vous voulez, tant que ce n'est pas du nom d'une fonction PHP existante. Dans le doute, gardez ce nom.

Ensuite, le code suivant en **bash** :

```
read var;  
echo var
```

⁴ Ainsi qu'un **STDERR**, que nous ne verrons pas ici.

devient ce code-ci en **PHP** :

```
$var = read();  
echo $var;
```

Testez-le dans un script ou dans le Shell Interactif **PHP**.

Notez au passage que vous venez d'apprendre comment on crée une fonction en PHP, mais c'est hors sujet pour ce cours. Vous verrez cela en Web Côté Serveur.

Tubes avec PHP

Pour terminer, maintenant que vous savez lire au clavier, ou plus précisément sur le **STDIN** du script, voici un exemple de script qui agit comme un filtre (**cut**, **sort** etc.).

Saisissez ce script **capital.php**, qui lit tout ce qui arrive sur son **STDIN**, jusqu'à ce qu'il n'y ait plus rien, et pour chaque ligne lue, il affiche chaque mot de la ligne avec une majuscule en premier et des minuscules sur les autres lettres. La fonction PHP `ucwords()` fait exactement ça. En PHP, il y a des fonctions pour tout, ou presque !

```
function read() {  
    static $stdin = null;  
  
    if ($stdin === null) {  
        $stdin = fopen('php://stdin', 'r');  
    }  
    return rtrim(fgets($stdin));  
}  
  
while ($line = read()) {  
    echo ucwords($line) . "\n";  
}
```

Testez là de cette façon sur le fichier **lang** :

```
./capital < lang
```

ou encore comme ceci sur les lignes contenant **fra** du fichier **prod** :


```
| grep fra < prod | tr ':' ' ' | ./capital
```

Vous devez comprendre cet enchaînement de filtres. Comme vous le voyez, il est très facile de créer des filtres avec du PHP pour compléter les besoins non couverts par les filtres standards.

Il est important de comprendre le code du script écrit **en rouge**.

Dans les ateliers de départ de ce TP, nous vous avons fait lire et stocker dans un tableau, avec la fonction **file()**, l'intégralité d'un fichier à traiter (**murphy**, **depts**, etc.). La raison était de vous simplifier la vie et de ne pas avoir à vous soucier de la façon dont on manipule des fichiers. En principe, vous devez apprécier cette simplicité. 😊

Avec la lecture du STDIN, c'est un peu différent et une petite explication s'impose donc. Décortiquons les deux lignes utiles du code :

```
| while ($line = read()) {
```

read() lit sur **STDIN** une ligne complète (complète signifie jusqu'au **\n**) et affecte la variable **\$line** avec ce qui a été lu.

S'il n'y a plus rien à lire sur **STDIN** un **read()** retourne **false**. La souplesse de PHP est encore présente ici : ses fonctions peuvent retourner des types différents. Exemple ici : une chaîne quand la lecture a pu se faire et un booléen (**false**) sinon.

Ainsi, le **while (...)** boucle tant que **read()** lit quelque chose et s'arrête sinon. Et pour chaque ligne lue :

```
|     echo ucwords($line) . "\n";
```

affiche une version *capitalisée* (ce que retourne la fonction PHP **ucwords()**) de ce qui a été lu + un **\n** pour revenir à la ligne.

Et voilà, c'est pas beau ?!

Q.99 Reprenez la Q.8 (donc le script 7) pour ne plus lire le fichier **access.log** dans le script mais en le passant par redirection entrante au script 7.