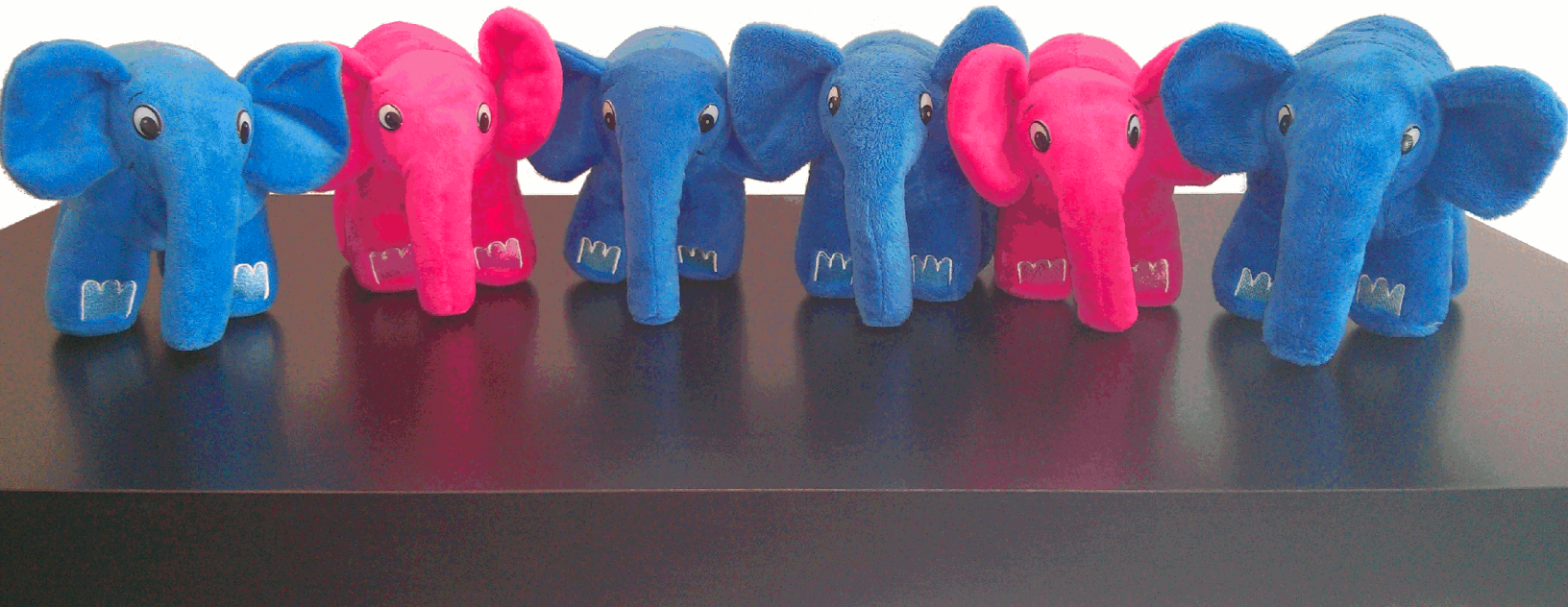
Pour PHP 5.3 à 5.6

Développer une **Extension PHP**

Pascal MARTIN



Développer une Extension PHP

Pascal MARTIN

This book is for sale at http://leanpub.com/developper-une-extension-php

This version was published on 2016-04-28

Tweet This Book !

Please help Pascal MARTIN by spreading the word about this book on Twitter !

The suggested tweet for this book is :

J'ai acheté le livre « Développer une extension PHP », par @pascal_martin — https ://leanpub.com/developper-une-extension-php

The suggested hashtag for this book is #extensionphp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter :

https ://twitter.com/search ?q=#extensionphp

Table des matières

À propos de cet aperçu

Ceci est un *aperçu* de mon livre **Développer une Extension PHP**, que vous pouvez acheter en version complète à l'adresse suivante :

> https ://leanpub.com/developper-une-extension-php[1]

Ce livre est disponible aux formats électroniques PDF, EPUB et MOBI.

Qu'est-ce que contient cet aperçu ?

Cet aperçu reproduit quelques extraits du livre complet :

- la préface,
- l'introduction,
- le premier chapitre, Une toute première extension, complet,
- et quelques petites sections plus indépendantes – pour l'instant regroupées en fin de livre.

J'espère que ces quelques dizaines de pages vous donneront envie d'en lire plus, tout en vous permettant de vérifier que le livre complet répondra à vos attentes !

Contenu du livre ?

Le livre complet fait plus de 520 pages et regroupe des chapitres traitant des sujets suivants :

- **Création d'une première extension** : il s'agit du premier chapitre du livre, qui est reproduit intégralement dans cet aperçu. J'y explique comment créer le squelette d'une première extension, quels fichiers sources sont requis et quelles structures de données et déclarations sont nécessaires.
- **Environnement de développement** : ce chapitre montre comment compiler une version de PHP orientée *développement d'extensions* sous Linux et quelles sont les informations qu'elle nous apporte. J'en ai profité pour présenter comment configurer Eclipse CDT pour obtenir un environnement de développement avec débogueur graphique.

[1] https://leanpub.com/developper-une-extension-php

- **Développer une extension PHP sous Windows** : ce chapitre fait suite au précédent, en expliquant comment mettre en place un environnement de développement avec Visual Studio, sous Windows, en allant de la compilation de PHP jusqu'au débogage en interface graphique.
- **Écrire une fonction** : les fonctions sont au cœur de PHP et de ses extensions. Nous verrons ici, entre autres, comment recevoir des paramètres et retourner une valeur. Ce sujet est abordé au travers de deux chapitres, le second intégrant des concepts supplémentaires, comme la réception de zval en paramètres, les fonctions attendant un nombre d'arguments variable, ou encore le type-hinting.
- **`zval` : les variables de PHP** : les variables de PHP sont représentées, en interne, par une structure nommée `zval`. Ce chapitre nous montrera comment en créer, les lire et les manipuler.
- **Gestion de la mémoire** : ce chapitre introduit le *Zend Memory Manager*, les fonctions d'allocation et de libération de mémoire, et présente comment travailler avec le TSRMLS.
- **`HashTable` et tableaux** : la structure `HashTable` est utilisée par PHP pour stocker un ensemble de données, comme un tableau. Elle est tellement importante pour PHP qu'elle est fréquemment utilisée en interne et que de nombreuses fonctions permettent de la manipuler.
- **Classes et objets** : PHP 5 a apporté un véritable modèle objet. Ce sujet est couvert à travers deux chapitres :
  - Le premier montre comment créer des classes exposant constantes, propriétés et méthodes, comment travailler avec de l'héritage ou des interfaces et comment stocker des données complexes.
  - Le second explique comment travailler avec des espaces de noms et des traits, présente quelques interfaces communément manipulées et passe en revue les gestionnaires d'objets les plus utilisés.
- **Erreurs et Exceptions** : PHP propose deux mécanismes de remontée d'*imprévus* à l'utilisateur : les erreurs, et les exceptions. Nous verrons ici comment tirer parti de ces deux principes, mais aussi comment transformer des erreurs internes en exceptions.
- **Configuration par fichier `.ini`** : le fichier `php.ini` représente, pour PHP et pour ses extensions, le standard de configuration permettant aux utilisateurs d'influencer sur leur comportement. Ce chapitre présentera comment exploiter au mieux cette possibilité de paramétrage.
- **Tests automatisés** : vous n'envisageriez pas de développer une extension sans tests automatisés ? Moi non plus !
- **Travailler avec des flux** : ce chapitre montre comment utiliser le mécanisme de flux de PHP, en lecture et en écriture. Il enchaîne ensuite avec la mise en place d'un gestionnaire de flux, avant de passer à la notion de contexte de flux et au développement d'un filtre de flux.
- **Les ressources** : nous verrons ici ce qu'est le type *ressource* et comment déclarer et manipuler des ressources, qu'elles soient ou non persistantes.
- **Configuration plus avancée de la compilation et du chargement de l'extension** : ce chapitre vous présentera comment mettre en place des dépendances entre extensions, comment utiliser une bibliothèque partagée externe, ou encore comment compiler votre extension de manière statique, intégrée à PHP.

- **Quelques points divers** : ce chapitre regroupe quelques points intéressants, mais qui ne méritaient pas à un chapitre à eux seuls, comme la déclaration de variables super-globales, de constantes, la personnalisation de `phpinfo()`, ou encore l'exécution de code PHP et l'utilisation de fonctions de rappel depuis une extension.
- **Les Extensions Zend** : ce chapitre, rédigé par Julien Pauli, montre de quelles possibilités dispose une extension Zend et comment en mettre en place – ainsi qu'un exemple d'une extension qui soit à la fois une extension PHP traditionnelle et une extension Zend.
- **Un œil dans la Zend Virtual Machine** : ce chapitre, rédigé par Julien Pauli, constitue une introduction au fonctionnement de la machine virtuelle de PHP et montre comment une `extension` ou une `zend_extension` peuvent *jouer* avec celui-ci.

Pour ce qui est des annexes, j'en ai mis une en place :

- **Normes de codage et bonnes pratiques PECL** : cette annexe reprend une partie des normes de codage de PHP et une série de bonnes pratiques issues de revues effectuées par l'équipe de PECL sur un nombre important d'extensions, traduites en français, filtrées et reformulées, et parfois accompagnées d'exemples supplémentaires.

La version publiée du livre sera bien sûr mise à jour, gratuitement, en cas d'éventuelles corrections.

Notez aussi que, aujourd'hui, aucune des plus de 75 extensions PHP écrites pour rédiger ce livre ne sont rendues publiques. Je n'ai pas encore vraiment réfléchi à l'idée, mais j'aimerais, à terme, publier au moins une partie d'entre-elles : cela constituerait un bon accompagnement aux différents chapitres. Cela dit, j'ai du travail à faire pour les rendre réellement *présentables* (ne serait-ce que pour les documenter et commenter leur code), et ce n'est donc pas pour tout de suite. Mais je garde l'idée à l'esprit !

Préface

Tout le monde connaît PHP aujourd'hui, c'est un fait. Enfin… connaît de l'extérieur, pour l'avoir utilisé du point de vue d'un développeur ou d'un administrateur.

Cet ouvrage, en revanche, est dédié aux entrailles du langage.

Ce que beaucoup d'utilisateurs de PHP ignorent, c'est que le langage est lui-même un programme, écrit avec le langage C, qui à la date où ces lignes sont écrites, comporte environ 700.000 lignes.
D'une complexité relativement élevée, la source de PHP témoigne de son fonctionnement le plus profond. Cependant, celle-ci n'est que très peu documentée, et se plonger dedans, même avec un passé en C costaud, n'est pas tâche facile.
S'intéresser à l'API interne de PHP permet, principalement au moyen de l'écriture d'extensions, de comprendre le fonctionnement du langage, de l'enrichir, d'en changer le fonctionnement ou encore de porter des parties de code PHP en langage C, beaucoup plus efficace et rapide en terme de traitement.

Le but de cet ouvrage est justement de vous guider pas à pas au travers de l'API de PHP. Vous découvrirez ainsi sa richesse, mais aussi sa complexité et son long historique (les premières lignes datent de 2000) qui font qu'elle est difficilement apprivoisable sans être tenu par la main.

Après des rappels sur la compilation de projets C sous Linux comme sous Windows, vous apprendrez à manipuler les structures indispensables du cœur de PHP. Puis, au travers de l'écriture d'une extension, vous verrez où et comment allouer et libérer de la mémoire, créer des fonctions PHP, jouer avec les classes et les objets, se brancher sur le système de configuration de PHP, manipuler les tableaux PHP de l'intérieur sans oublier la maîtrise du curieux type « ressource ». Tout ceci sera complété de conseils, de bonnes pratiques à suivre et de pièges à éviter.

Enfin, une introduction aux pièces les plus complexes vous permettra de découvrir sereinement les concepts de compilation de code, d'OPCode, de caches ou encore de machine virtuelle.

Je vous souhaite une bonne lecture.

– **Julien Pauli**
Contributeur PHP et release manager de PHP 5.5 & 5.6

Introduction

> *There is no programming language, no matter how structured, that will prevent programmers from making bad programs.*
>
> – **Larry Flon**

Une Extension PHP est un module chargé par le moteur de PHP lors de son lancement et capable d'influer sur son comportement, généralement en ajoutant des fonctions ou classes qui sont alors considérées comme *internes* à PHP.

Pourquoi développer une extension PHP ?

Plusieurs raisons peuvent justifier le choix d'écrire une extension PHP. Pour n'en citer que quelques-unes :

- une extension est écrite en C, qui est un langage compilé de bas niveau, ce qui permet souvent un gain en performances considérable par rapport à une portion de code équivalente écrite en PHP.
- Une extension C peut être utilisée pour *encapsuler* une bibliothèque système, l'exposant ainsi à l'espace utilisateur PHP.
- De par son intégration plus proche du cœur de PHP, une extension a des possibilités fonctionnelles qui ne sont pas offertes aux scripts utilisateurs.
- Ce point n'est que rarement exploité et assez éloigné de la philosophie ouverte de PHP, mais diffuser la version compilée d'une extension sans son code source permet de mettre en place un composant *boite-noire*, exposant des fonctionnalités aux utilisateurs sans pour autant leur révéler comment celles-ci sont implémentées.

Vous l'aurez compris : à partir du moment où vous développez une extension PHP, un large éventail de possibilités s'offrent à vous – jetez un coup d'œil à la liste d'extensions diffusées sur PECL[2] pour vous en rendre compte par vous-même !

Pour donner quelques exemples, correspondant respectivement à chacune de ces raisons :

- le moteur de templating Twig[3] propose une extension, plus rapide que sa version PHP.

[2] http://pecl.php.net/packages.php
[3] http://twig.sensiolabs.org/

- L'extension `ssh2`[4] permet à PHP d'utiliser les fonctionnalités de la bibliothèque `libssh2`[5], de la même manière que l'extension `curl`[6] encapsule la bibliothèque `libcurl`[7].
- Une extension peut déclarer des variables super-globales, comme le fait `ext/session`[8] pour `$_SESSION`. Une extension PHP peut exposer à l'espace utilisateur le mécanisme de threading intégré à PHP, comme le fait `pthread`[9]. Une extension PHP, comme `AOP`[10] peut aussi permettre de se brancher autour de l'exécution de n'importe quelle fonction.
- L'extension `ioncube`[11] permet de lire à la volée des fichiers PHP encodés avant leur diffusion. Un autre exemple serait l'extension PHP que newrelic[12] fournit, qui permet d'envoyer à son service de monitoring et d'analyse des données internes sur le fonctionnement de votre application. Ces extensions sont toutes deux propriétaires et leurs sources ne sont pas diffusées.

Bref, les possibilités sont pour ainsi dire infinies !

Pourquoi ne pas développer une extension PHP ?

Pour autant, développer une extension PHP n'est pas la solution à tous vos problèmes et trois points majeurs sont à prendre en compte avant de vous lancer.

Le premier est qu'une extension s'écrit en C. Ce langage n'est pas, en soi, plus *difficile* qu'un autre, mais si vous travaillez dans un contexte plutôt orienté Web, il est fort probable que vos collègues (ou vous-même !) ne soient pas à l'aise avec ce langage. Cela rendra le développement et la maintenance de votre extension plus difficile : qui s'en chargera si vous êtes en vacances[13] ?

Le second est qu'une extension s'intègre plus profondément au moteur de PHP qu'un script utilisateur. En conséquence, un bug au sein d'une extension peut avoir des conséquences nettement plus dramatiques : au lieu d'un avertissement ou, au pire, d'une Fatal Error mettant fin à l'exécution d'une page, vous pouvez causer un plantage du serveur Web !

Le troisième est un peu lié au second : à partir du moment où l'on travaille au plus proche de l'*interne* du moteur de PHP, on ressent beaucoup plus les modifications apportées à celui-ci (ce qui est logique). Les développeurs de PHP s'assurent que les améliorations apportées à celui-ci aient le moins d'impact possible au niveau de l'exécution de scripts PHP, mais on ne peut pas toujours en dire autant lorsqu'il s'agit des extensions. Autrement dit, développer une extension proche du moteur de PHP – *ce qui n'est pas le cas de toute, heureusement* – qui fonctionne sous PHP 5.3 et 5.4

[4] http://php.net/book.ssh2
[5] http://www.libssh2.org/
[6] http://php.net/book.curl
[7] http://curl.haxx.se/libcurl/
[8] http://php.net/book.session
[9] http://php.net/book.pthreads
[10] http://pecl.php.net/package/AOP
[11] http://www.ioncube.com/loaders.php
[12] http://newrelic.com/
[13] Et si vous *passez sous un bus* ?

et 5.5 voire même 5.6 n'est pas une tâche facile sans une parfaite connaissance de ces changements, qui ne sont pas toujours réellement perceptibles de l'extérieur.

Bien sûr, il vous revient de faire le nécessaire pour limiter les risques : comme pour n'importe quel composant, partagez le savoir autour de vous ; et mettez en place des tests automatisés !

Enfin, installer une extension PHP demande généralement l'intervention d'un administrateur. Le déploiement d'une mise à jour de celle-ci est donc souvent moins évident que lorsqu'il s'agit de livrer du code PHP.

À qui s'adresse ce livre ?

Ce livre s'adresse à des développeurs PHP expérimentés et suppose que vous connaissez suffisamment bien PHP pour savoir ce qu'il permet, ce qu'il ne permet pas, et vouloir aller plus loin.

De bonnes notions de C, même un peu *rouillées* et/ou remontant par exemple à vos études, seraient un plus non négligeable : elles vous permettraient de comprendre plus facilement une partie des constructions utilisées. Toutefois, si vous n'avez pas peur d'apprendre *sur le tas* et que vous êtes prêt à fouiller par vous-même, les points de C utilisés au cours de ce livre devraient rester abordables même pour un débutant.

Les exemples présentés ont tous été testés sur PHP 5.4[14]. La majeure partie d'entre eux devraient toutefois fonctionner sur des versions inférieures (PHP 5.3, voire même l'obsolète version 5.2), mais j'aborderai parfois – en indiquant lorsque ce sera le cas – des concepts qui ne peuvent être exploités qu'à partir de PHP 5.5.

Remerciements

Un immense merci à Julien Pauli (@julienPauli[15]), Release Manager de PHP 5.5 et 5.6, pour les nombreux retours techniques qu'il m'a adressés en relisant ce livre, qui se sont traduits par autant de corrections ou d'ajouts de précisions.

Les sections en rapport avec le travail sous Windows doivent beaucoup à Pierre Joye (@pierrejoye[16]), qui contribue à PHP depuis de nombreuses années, à qui je dois également un grand merci !

Je tiens aussi à remercier Agnès Haasser[17] (@tut_tuuut[18]) qui a accepté de relire plusieurs chapitres de ce livre, corrigeant de nombreuses fautes de français ici et là, rendant ainsi la lecture plus agréable pour tous.

[14] Pendant le plus gros du temps passé à de l'écriture de ce livre, PHP 5.4 est la version stable de PHP, PHP 5.3 atteignant sa fin de vie et PHP 5.5 ayant été diffusé il y a peu de temps.

[15] https://twitter.com/julienPauli

[16] https://twitter.com/pierrejoye

[17] http://www.ploque.net/

[18] https://twitter.com/tut_tuuut

Enfin, merci aussi à vous qui prenez le temps de me signaler[19] les inévitables fautes, erreurs, ou même bugs que vous constatez : la qualité de ce livre en est d'autant améliorée. Continuez ! Je suis aussi bien évidemment ouvert aux suggestions et recommandations que vous avez à l'esprit. ;-)

À propos de l'auteur

Je m'appelle Pascal MARTIN.

J'ai découvert PHP aux environs de l'an 2000 et je travaille dans le développement Web et PHP depuis plus de 9 ans. Je publie occasionnellement des articles en rapport avec le développement Web et principalement PHP, sur mon blog[20].

Ayant de plus en plus tendance, depuis quelques années, à chercher comment PHP fonctionne en lisant des portions de son code source, c'est presque naturellement que j'en suis venu à me pencher sur le développement d'extensions, renouant ainsi avec le C, langage qui m'a réellement fait découvrir la programmation il y a bientôt 15 ans.

Vous pouvez me suivre sur @pascal_martin[21] et parfois me croiser sur Google+[22] ou sur StackOverflow[23], ou me joindre par e-mail : contact@pascal-martin.fr[24].

Je suis bien sûr preneur de tout retour, de toute suggestion et de tout rapport d'erreur que vous voudrez me faire ;-)

> *Je vous souhaite une excellente lecture et bon courage pour vos premiers pas dans le développement d'extensions PHP !*
>
> – **Pascal MARTIN**

[19] mailto:contact@pascal-martin.fr
[20] http://blog.pascal-martin.fr/
[21] https://twitter.com/pascal_martin
[22] https://plus.google.com/100626033023167917165?rel=author
[23] http://stackoverflow.com/users/138475/pascal-martin
[24] mailto:contact@pascal-martin.fr

1. Une toute première extension

> *The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.*
>
> – **Edsger W. Dijkstra**

Créer un squelette d'extension PHP, qui ne fait *rien*, mais qui est reconnue en tant que telle et chargée par le moteur, demande de créer quelques fichiers : un fichier de code source C, un fichier d'en-têtes et un fichier permettant de configurer la compilation de l'extension.

1.1 Fichiers sources et configuration de la compilation

Dans sa version la plus simple qui soit, une extension PHP n'est composée que de quelques fichiers :

- un fichier `.c`, qui contiendra le code source de l'extension,
- un fichier `.h`, qui, à terme, hébergera les définitions de types et de fonctions,
- et un fichier `config.m4` qui configure la compilation sous les systèmes UNIX.

En complément, on trouve aussi, généralement :

- quatre fichiers texte nommés `README`, `CREDITS`, `API-version`, et `RELEASE-version`,
- un fichier `LICENSE` ou `COPYING` indiquant sous quelle licence l'extension est distribuée,
- et un fichier `config.w32`, qui configure la compilation sous Windows.

1.1.1 Fichiers sources de l'extension

Par convention, le fichier qui contiendra le code source de votre extension (le fichier « principal », tout au moins, si vous en avez plusieurs) porte le nom de l'extension ; autrement dit, si vous développez une extension nommée `monext01`, alors, son fichier source principal sera nommé `monext01.c`.

Pour une extension la plus basique qui soit, le code source que vous devrez mettre en place aura la forme suivante :

```c
#include "php_monext.h"

/* {{{ monext_module_entry */
zend_module_entry monext_module_entry = {
    STANDARD_MODULE_HEADER,
    "monext",
    NULL, /* Function entries */
    NULL, /* Module init */
    NULL, /* Module shutdown */
    NULL, /* Request init */
    NULL, /* Request shutdown */
    NULL, /* Module information */
    "0.1", /* Replace with version number for your extension */
    STANDARD_MODULE_PROPERTIES
};
/* }}} */

#ifdef COMPILE_DL_MONEXT
ZEND_GET_MODULE(monext)
#endif
```

Ce fichier `monext01.c` est composé de trois sections :

- tout d'abord, le contenu du fichier `php_monext.h`, que nous verrons juste en dessous, est inclus ; exactement comme avec la directive `include`[1] de PHP, cela revient à copier-coller le contenu du fichier vers l'endroit où la directive est écrite.
- Ensuite, nous renseignons une structure de type `zend_module_entry`, dont le nom est `_module_entry` précédé du nom de notre extension : `monext_module_entry` ; cette structure permet au moteur de PHP de charger notre extension.
- Et enfin, la dernière section fait appel à `ZEND_GET_MODULE()` lorsque notre extension est chargée dynamiquement ; c'est cette ligne qui permettra au moteur de PHP de découvrir la structure que nous avons définie juste au-dessus.

Pour l'instant, nous n'avons renseigné que peu de champs de la structure `zend_module_entry` de notre extension : le nom et la version. Nous verrons au cours des prochains chapitres de ce livre que cette structure permet de définir la liste des fonctions exportées par notre extension, des fonctions exécutées lors de son chargement et déchargement…

Le fichier `php_monext.h` auquel faisait référence `monext01.c` aura, quant à lui, la forme suivante :

[1] http://php.net/function.include

```
#ifndef MONEXT_H_
#define MONEXT_H_

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"

#endif /* MONEXT_H_ */
```

Pour l'instant, considérant la simplicité de notre extension, ce fichier d'en-têtes ne fait rien de plus qu'inclure quelques fichiers fournis par PHP [2], dont, en particulier, le fichier d'en-têtes `php.h`.

Ici, le nom de l'extension et sa version ont été renseignés *en dur* dans la structure `monext_module_entry`. À terme, vous aurez généralement tendance à définir depuis le fichier `.h` de votre extension une constante nommée `PHP_nom_de_lextension_VERSION` pour définir son numéro de version :

```
#define PHP_MONEXT_VERSION "0.1"
```

Cette constante sera alors utilisée dans la définition de la structure déclarant l'extension au moteur de PHP :

```
zend_module_entry monext_module_entry = {
    STANDARD_MODULE_HEADER,
    "monext",
    NULL, /* Function entries */
    NULL, /* Module init */
    NULL, /* Module shutdown */
    NULL, /* Request init */
    NULL, /* Request shutdown */
    NULL, /* Module information */
    PHP_MONEXT_VERSION, /* Replace with version number for your extension */
    STANDARD_MODULE_PROPERTIES
};
```

Si vous êtes amené à distribuer cette extension via PECL, cette constante sera utilisée pour déterminer le numéro de version de l'extension indiqué sur le site.

Au cours des prochains chapitres, au fur et à mesure de l'ajout de fonctionnalités à notre extension, nous viendrons enrichir ce fichier.

[2] En fait, nous aurions pu placer ces directives d'inclusion directement dans `monext.c` et ne pas du tout utiliser de fichier `.h` ; mais, puisque nous utiliserons réellement ce fichier par la suite, autant prendre dès maintenant la bonne habitude de le définir.

1.1.2 Configuration de la compilation

Une fois les deux fichiers sources de l'extension en place, il faut configurer la compilation de celle-ci. Sous un système UNIX-like, cela se fait via un fichier nommé `config.m4`.

Ce fichier est responsable de la création d'une option qui permettra d'activer la compilation de l'extension lors de l'appel à `./configure`, ainsi que de la déclaration du module et de l'ensemble des fichiers source `.c` qui le composent.

Voici le fichier `config.m4` que nous pouvons utiliser pour configurer la compilation de notre première extension :

```
PHP_ARG_ENABLE(monext, whether to enable monext support,
[  --enable-monext         Enable monext support])

if test "$PHP_MONEXT" = "yes"; then
    PHP_NEW_EXTENSION(monext, monext.c, $ext_shared)
fi
```

La première portion de ce fichier utilise `PHP_ARG_ENABLE()` pour ajouter une option, `--enable-monext`, au script de configuration ; cette option pourra être utilisée lors de l'exécution de `./configure`, pour activer la compilation de l'extension.

En fonction du nom de l'extension, il vous faudra souvent adapter le nombre d'espaces entre l'option permettant d'activer sa compilation et la description de celle-ci, pour que la sortie de `./configure --help` soit correctement alignée.

Le dernier paramètre passé à cette option se retrouve dans la sortie de `./configure --help` pour indiquer l'utilité de l'option correspondante :

```
./configure --help
`configure' configures this package to adapt to many kinds of systems.
...
Optional Features and Packages:
...
  --enable-monext         Enable monext support
...
```

La seconde partie du fichier `config.m4` que nous avons créé ici utilise l'instruction `PHP_NEW_EXTENSION()` pour déclarer une extension PHP, dans le cas où sa compilation a été demandée via l'option `--enable-monext`.

`PHP_NEW_EXTENSION()` attend en second paramètre la liste de l'ensemble des fichiers `.c` qui composent l'extension : nous n'en avons qu'un seul pour l'instant, mais si nous sommes amenés à en ajouter, il faudra les déclarer ici.

Nous verrons plus loin dans ce livre comment mettre en place le fichier `config.w32`, qui joue le même rôle lorsqu'il s'agit de compiler une extension sous Windows.

1.2 Compilation de l'extension

Une fois que nous avons écrit le code source de notre extension et le fichier configurant sa compilation, cette compilation se fait en trois étapes.

Pour ce chapitre, nous compilons notre extension pour la version de PHP installée au niveau *système*. Nous verrons plus loin comment compiler une extension pour une version spécifique de PHP (qu'il s'agisse de cibler un numéro de version particulier ou une version de PHP compilée avec informations de débogage).

La première étape de la compilation d'une extension est de lancer la commande `phpize` :

```
$ phpize
Configuring for:
PHP Api Version:         20100412
Zend Module Api No:      20100525
Zend Extension Api No:   220100525
```

L'utilitaire `phpize` va créer le script `configure` à partir des informations présentes dans le fichier `config.m4` et de la configuration de PHP (version de PHP – qui se retrouve dans les numéros de versions affichés en sortie de `phpize` – mais aussi options qui avaient été utilisées lors de sa compilation, comme activation ou non du débogage).

La commande `phpize` extrayant les informations de `config.m4` pour créer le script `configure`, elle sera à relancer à chaque modification du fichier de configuration `config.m4`.

Après cela, il devient possible d'exécuter le script `./configure` qui vient d'être créé :

```
$ ./configure --enable-monext
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for a sed that does not truncate output... /bin/sed
[...]
[...]
checking whether to build static libraries... no
configure: creating ./config.status
config.status: creating config.h
config.status: executing libtool commands
```

Le script `configure` vérifie que les outils et bibliothèques requis pour construire l'extension sont tous présents sur votre système et crée le fichier `Makefile` qui permettra réellement de compiler celle-ci.

Si le script `./configure` échoue, c'est généralement parce qu'il vous manque une bibliothèque ; le message d'erreur affiché devrait vous permettre de déterminer laquelle – installez-la en version de développement (sous distributions debian, il s'agit généralement d'un paquet nommé comme la bibliothèque, mais avec un suffixe `-dev`).

Une fois l'étape de configuration passée avec succès, vous pouvez exécuter la commande `make` pour lancer la compilation de l'extension :

```
$ make
[...]
[...]
----------------------------------------------------------------------
Libraries have been installed in:
   /home/squale/developpement/book-php-extension/sources/monext01/modules

[...]
[...]
----------------------------------------------------------------------

Build complete.
Don't forget to run 'make test'.
```

Comme l'indique la sortie de `make`, l'extension est compilée vers le répertoire `modules/` :

```
$ ls -1 modules
monext.la
monext.so
```

Notre extension est le fichier `monext.so`.

1.3 Et voila, une nouvelle extension !

Puisque notre extension est compilée, il ne nous reste plus qu'à utiliser la directive de configuration `extension` de PHP pour la charger.

1.3.1 Vérifier que l'extension est chargée

En ligne de commande, en exploitant l'option `-m` de PHP, nous pouvons vérifier que notre nouvelle extension est reconnue par PHP :

```
$ php -dextension=modules/monext.so -m
[PHP Modules]
apc
bcmath
bz2
[...]
monext
[...]
zip
zlib
```

Lorsque nous indiquons à PHP qu'il doit charger le fichier `monext.so` de notre extension, nous pouvons voir que celle-ci fait partie des extensions chargées ; et aucun message d'erreur n'est affiché.

Bien sûr, en l'état, notre extension ne fait rien ; mais maintenant que nous avons un squelette d'extension *vide*, nous allons pouvoir, petit à petit, lui ajouter les fonctionnalités qui nous intéressent !

1.3.2 Et `phpinfo()` alors ?

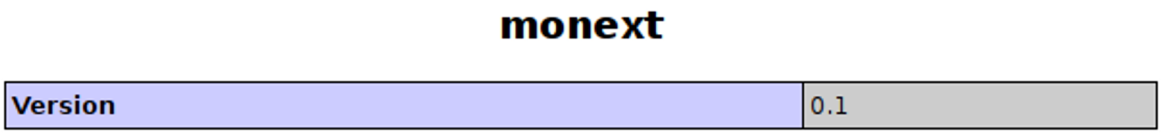
À présent, créons un fichier `phpinfo.php` contenant un appel à la fonction `phpinfo()` de PHP, comme ceci :

```php
<?php
phpinfo();
?>
```

Si nous avons compilé notre extension avec PHP 5.4 ou supérieur, nous pouvons tirer parti du serveur web de test intégré, pour servir ce fichier PHP sans avoir à reconfigurer quoi que soit pour que PHP soit exécuté par un serveur Web distinct (Apache, nginx…) que vous pourriez avoir sur votre machine :

```
$ php -dextension=modules/monext.so -S localhost:8080
PHP 5.4.6-1ubuntu1.1 Development Server started at Sun Mar 10 17:36:48 2013
Listening on http://localhost:8080
Document root is /.../monext01
Press Ctrl-C to quit.
[Sun Mar 10 17:36:54 2013] 127.0.0.1:53260 [200]: /phpinfo.php
```

Charger `http://localhost:8080/phpinfo.php` dans votre navigateur affichera la sortie de `phpinfo()`, incluant une section – minimaliste – à propos de notre extension :



La sortie de `phpinfo()` pour notre extension

Comme nous pouvons le constater, par défaut, un squelette d'entrée a été automatiquement généré pour `phpinfo()`. Nous verrons dans les prochains chapitres comment enrichir celui-ci pour y faire remonter plus d'informations à propos de notre extension.

1.4 Contrôle de sources et fichiers à ignorer

Si vous regardez le contenu du répertoire au sein duquel vous avez travaillé pour ce chapitre, vous verrez que là où vous n'aviez créé que quelques fichiers (un fichier `.c`, un fichier `.h`, le fichier `config.m4`, et éventuellement un fichier `.php`), vous en avez maintenant plus d'une vingtaine, y compris quelques répertoires.

En effet, le processus de compilation d'une extension PHP crée un nombre conséquent de fichiers – fichiers qui ne font pas en soi partie du code de votre extension et ne devraient pas être commités sur votre gestionnaire de code source, puisqu'ils ont été automatiquement générés, peuvent être recréés au besoin, et, en plus de cela, dépendent pour certains de votre environnement.

Vous voudrez sans aucun doute indiquer à votre gestionnaire de code source[3] qu'il doit ignorer le gros de ces fichiers. Si vous travaillez avec Git, cela peut être fait en créant à la racine de votre projet un fichier `.gitignore` contenant la liste des motifs de fichiers à exclure ; si vous travaillez avec SVN, vous pouvez arriver au même résultat en utilisant la propriété `svn:ignore`. Dans tous les cas, la liste de fichiers que vous souhaiterez exclure ressemblera à celle-ci :

```
.deps
*.lo
*.la
.libs
Makefile
Makefile.fragments
Makefile.global
Makefile.objects
*.tgz
acinclude.m4
aclocal.m4
build
config.cache
config.guess
config.h
config.h.in
config.log
config.nice
config.status
config.sub
configure
configure.in
conftest
conftest.c
include
install-sh
libtool
ltmain.sh
missing
mkinstalldirs
modules
sm.php
run-tests.php
autom4te.cache
```

[3] Bien sûr, comme pour n'importe quel autre projet, vous avez prévu de stocker les sources de votre extension PHP au sein d'un gestionnaire de code source, comme Git ou Subversion. Si non… Eh bien, il n'est jamais trop tard pour bien faire. ;-)

Avec cela, les seuls fichiers qui seront placés sous gestion de code seront les vrais fichiers sources de notre extension : ceux que nous avons créés au cours de ce chapitre.

En complément, notez que l'option `--clean` du script `phpize` permet d'effacer tous les fichiers temporaires qui sont créés lors des différentes étapes de la compilation d'une extension PHP, ainsi que ceux laissés lorsque des tests automatisés échouent.

2. Quelques points divers

Ce dernier *chapitre*, qui n'en est en fait pas vraiment un, est destiné à regrouper quelques sujets distincts les uns des autres, qui ne se seraient pas en l'état actuel intégrés dans d'autres chapitres de ce livre et ne sont pas assez longs pour constituer des chapitres à part entière, mais dont je tenais à parler – et qui méritent mieux qu'un classement en tant qu'*annexe*.

Il est possible que les sections qui figurent ici dans la version courante de ce livre soient revues, peut-être en profondeur, dans une prochaine version. Il est même possible que certaines soient intégrées à un autre chapitre, ou disparaissent complètement.

2.1 Déclarer des constantes

Il est possible de définir des constantes depuis du code PHP utilisateur en utilisant l'instruction `define()`[1]. Bien entendu, une extension peut elle aussi exposer des constantes.

2.1.1 Déclarer une constante

Une constante aura généralement toujours la même valeur – constante. Elle est donc définie, dans une extension PHP, au moment du chargement de celle-ci ; c'est-à-dire au moment de la phase `MINIT`, depuis la fonction branchée sur celle-ci.

Si une constante doit avoir une valeur qui peut être différente pour chaque requête (mais qui reste la même tout au long du traitement de chaque requête, bien sûr – ça ne serait pas une constante, sinon !), elle peut être définie depuis la fonction branchée sur la phase `RINIT` de votre extension.

Le plus souvent, une constante sera définie en utilisant une des macros `REGISTER_*_CONSTANT()` :

```
PHP_MINIT_FUNCTION(monext)
{
    REGISTER_STRING_CONSTANT("MONEXT_CTE_STR_1", "1ère constante",
        CONST_PERSISTENT);
    REGISTER_STRING_CONSTANT("MONEXT_CTE_STR_2", "2nde constante",
        CONST_CS | CONST_PERSISTENT);

    return SUCCESS;
}
```

[1] http://php.net/define

Pour définir une constante de type chaîne de caractères, ici, nous avons utilisé la macro `REGISTER_STRING_CONSTANT()`, qui attend trois paramètres :

- le nom de la constante, qui doit être unique, une constante ne pouvant être définie qu'une seule fois,
- la valeur de cette constante,
- et une combinaison de drapeaux.

Trois valeurs peuvent être combinées, pour ce dernier paramètre :

- `CONST_PERSISTENT` : indique que la constante doit être persistante, conservée d'une requête à l'autre. Cette valeur sera utilisée pour les constantes définies au moment de la phase `MINIT`, et pas pour celles qui seraient définies depuis la phase `RINIT`.
- `CONST_CS` : indique que le nom de la constante est sensible à la casse. En général, un nom de constante se trouve en majuscules et est sensible à la casse, et ce drapeau est donc utilisé. Au niveau des constantes déclarées par PHP, ce flag est activé pour toutes, sauf pour `TRUE`, `FALSE`, et `NULL`, dont les noms sont insensibles à la casse.
- `CONST_CT_SUBST` : cette valeur n'est que rarement utilisée et permet une optimisation (substitution lors de la compilation d'un script PHP) lorsque `CONST_CS` n'est pas utilisée.

À titre d'exemple, voici un script PHP utilisant les deux constantes définies un peu plus haut :

```
<?php
// Nom de constante insensible à la casse
var_dump(MONEXT_CTE_STR_1);
var_dump(MonExt_cTe_StR_1);

// Nom de constante sensible à la casse
var_dump(MONEXT_CTE_STR_2);

// Erreur
var_dump(MonExt_cTe_StR_2);
?>
```

En exécutant ce script, voici la sortie que nous obtiendrions :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
string(15) "1ère constante"
string(15) "1ère constante"
string(14) "2nde constante"
PHP Notice:  Use of undefined constant MonExt_cTe_StR_2
    - assumed 'MonExt_cTe_StR_2' in .../test.php on line 10

Notice: Use of undefined constant MonExt_cTe_StR_2
    - assumed 'MonExt_cTe_StR_2' in .../test.php on line 10
string(16) "MonExt_cTe_StR_2"
```

N'importe quelle combinaison de majuscules et minuscules peut être utilisée pour désigner notre première constante, dont le nom n'est pas sensible à la casse, alors que le nom de la seconde ne peut être écrit qu'en majuscules, puisqu'elle a été déclarée avec le flag `CONST_CS`.

PHP fournit les macros suivantes pour déclarer des constantes :

- `REGISTER_LONG_CONSTANT()` : permet de déclarer une constante entière,
- `REGISTER_DOUBLE_CONSTANT()` : pour définir une constante en tant que nombre à virgule à flottante,
- `REGISTER_STRING_CONSTANT()` : nous l'avons utilisée un peu plus haut : elle permet de définir une constante de type chaîne de caractères,
- `REGISTER_STRINGL_CONSTANT()` : cette dernière macro permet elle aussi de définir une constante chaîne de caractères, mais en spécifiant la longueur de cette chaîne de troisième paramètre, les drapeaux étant alors le quatrième paramètre.

2.1.2 Noms et valeurs dynamiques

Les macros vues juste au-dessus utilisent `sizeof()` pour déterminer la longueur de la chaîne de caractères correspondant au nom de la constante à définir, ce qui empêche de les utiliser avec un nom de constante qui ne soit pas écrit *en dur* dans le code, lors de leur appel : il n'est pas possible d'utiliser quelque chose de ce type :

```
char * nom = "...";
REGISTER_LONG_CONSTANT(nom, 123456, CONST_CS | CONST_PERSISTENT);
```

`sizeof(nom)` serait évalué comme le nombre d'octets utilisés pour représenter un pointeur `char *`, et pas comme le nombre de caractères du nom de la constante.

À la place, pour déclarer une constante dont le nom n'est pas connu à la compilation de l'extension, vous allez devoir directement faire appel aux fonctions qui sont finalement utilisées par ces macros : chaque macro correspond à une fonction nommée sous la forme `zend_register_*_constant()`.

Par exemple, pour définir cinq constantes, toutes de type chaîne de caractères, nommées de `MONEXT_CTE_STR_1` à `MONEXT_CTE_STR_5`, nous pourrions utiliser une portion de code ressemblant à celle-ci :

```
PHP_MINIT_FUNCTION(monext)
{
    int i;
    for (i=1 ; i<=5 ; i++)
    {
        char *nom;
        /* longueur = prefixe + longueur(i) + octet null de fin de chaine */
        nom = emalloc(strlen("MONEXT_CTE_STR_") + 1 + 1);
        char *valeur = pemalloc(strlen("constante n°") + 1 + 1, 1);

        sprintf(nom, "MONEXT_CTE_STR_%d", i);
        sprintf(valeur, "constante n°%d", i);
        zend_register_string_constant(
            nom,
            strlen(nom) + 1,
            valeur,
            CONST_CS | CONST_PERSISTENT,
            module_number TSRMLS_CC
        );

        efree(nom);
    }

    return SUCCESS;
}
```

Les fonctions `zend_register_*_constant()` s'utilisent de manière assez similaire à celle des macros correspondantes ; elles attendent quelques paramètres supplémentaires :

- un second paramètre est intercalé après le nom de la constante : la longueur de ce nom. Attention, le caractère nul de fin de chaîne doit être compté – d'où le `strlen() + 1` ici,
- le numéro de l'extension, reçu par toutes fonctions branchées sur les phases `MINIT` et `RINIT` lors de l'expansion des macros `PHP_MINIT_FUNCTION()` et `PHP_RINIT_FUNCTION()`, doit être passé après les drapeaux, pour que le moteur PHP sache rattacher la constante à notre extension.

Notez que lorsque vous déclarez une constante de type de chaîne de caractères, la valeur spécifiée n'est pas copiée vers ladite constante, mais uniquement référencée par celle-ci. Cela signifie que les chaînes de caractères créées dynamiquement, comme celles définies ici, doivent être allouées en mémoire permanente.

Utilisons le script `test.php` suivant pour tester l'affichage de nos cinq constantes créées avec des noms et valeurs dynamiques :

```
<?php
var_dump(MONEXT_CTE_STR_1);
var_dump(MONEXT_CTE_STR_2);
var_dump(MONEXT_CTE_STR_3);
var_dump(MONEXT_CTE_STR_4);
var_dump(MONEXT_CTE_STR_5);
?>
```

Exécuter ce script nous donnera la sortie reproduite ci-dessous :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
string(14) "constante n°1"
string(14) "constante n°2"
string(14) "constante n°3"
string(14) "constante n°4"
string(14) "constante n°5"
```

2.1.3 Constantes d'autres types

Les macros `REGISTER_*_CONSTANT()` et fonctions `zend_register_*_constant()` correspondantes que nous avons vues jusqu'à présent permettent de définir des constantes de types numériques entier et flottant et chaîne de caractères ; mais il est tout à fait possible de créer des constantes *manuellement* en construisant une variable de type `zend_constant`[2], qui est une structure définie comme suit :

```
typedef struct _zend_constant {
    zval value;
    int flags;
    char *name;
    uint name_len;
    int module_number;
} zend_constant;
```

Ensuite, à nous de renseigner ces champs un par un et d'utiliser la fonction `zend_register_constant()` pour enregistrer la constante auprès du moteur de PHP.

Par exemple, pour définir une constante de type booléen, nous pourrions utiliser une portion de code similaire à celle-ci :

[2] La structure `zend_constant` est définie dans le fichier `Zend/zend_constants.h`.

```
PHP_MINIT_FUNCTION(monext)
{
    /* Définition d'une constante booléenne */
    zend_constant c;
    c.flags = CONST_CS | CONST_PERSISTENT | CONST_CT_SUBST;
    c.module_number = module_number;
    c.name = zend_strndup(ZEND_STRL("MONEXT_CTE_BOOL"));
    c.name_len = sizeof("MONEXT_CTE_BOOL");
    c.value.value.lval = 1;
    c.value.type = IS_BOOL;
    zend_register_constant(&c TSRMLS_CC);

    return SUCCESS;
}
```

La constante est ensuite accessible, comme celles créées précédemment, depuis un script PHP :

```
<?php
var_dump(MONEXT_CTE_BOOL);
?>
```

Et l'exécution de ce script donne la sortie attendue :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
bool(true)
```

Les macros `REGISTER_NULL_CONSTANT()` et `REGISTER_BOOL_CONSTANT()`, ainsi que les fonctions correspondantes `zend_register_bool_constant()` et `zend_register_null_constant()`, n'existent pas en PHP 5.4 ni 5.5, mais il est possible qu'elles soient implémentées pour la version suivante[3].

2.1.4 Lire la valeur d'une constante

La valeur d'une constante peut être lue, depuis une extension PHP, en appelant la fonction `zend_get_constant()`, qui prend en paramètres le nom de celle-ci et un pointeur vers une `zval` qui permettra de stocker une copie de la valeur recherchée.

Par exemple, nous pourrions écrire la fonction suivante, qui prend elle-même en paramètre le nom d'une constante à lire et affiche la valeur correspondante :

[3] Ces deux macros et ces deux fonctions ont été commitées sur la branche `master` de PHP après la création de la branche correspondant à PHP 5.5. Il y a donc de bonnes chances qu'elles fassent leur apparition pour PHP 5.6 – et que vous ne puissiez pas les utiliser si votre extension est compilée avec une version inférieure de PHP.

```
PHP_FUNCTION(monext_get_constant)
{
    /* zval vers laquelle sera stockée la valeur de la constante */
    zval val;

    char *name;
    int name_len;

    /* Nom de la constante, passé en paramètre */
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
        &name, &name_len) == FAILURE) {
        return;
    }

    if (zend_get_constant(name, name_len, &val TSRMLS_CC))
    {
        /* Pour affichage, il est plus facile d'avoir une chaine de caractères */
        convert_to_string(&val);

        PUTS("Constante ");
        PHPWRITE(name, name_len);
        PUTS(" : ");
        PHPWRITE(Z_STRVAL(val), Z_STRLEN(val));
        PUTS("\n");

        zval_dtor(&val);
    }
}
```

Notez que, en vue de faciliter l'affichage, nous avons converti en chaîne de caractères la `zval` renseignée par `zend_get_constant()`. Cette `zval` correspondant à une copie de la valeur de la constante et non directement à cette valeur, nous pouvons la manipuler comme bon nous semble.

Notre fonction `monext_get_constant()` peut être appelée depuis PHP, pour afficher la valeur d'une constante définie par notre extension (cf plus haut), d'une constante définie depuis le script PHP, ou d'une constante inexistante :

```
define('PHP_CTE_DOUBLE_1', 3.1415);

monext_get_constant('MONEXT_CTE_STR_1');
monext_get_constant('PHP_CTE_DOUBLE_1');
monext_get_constant('CONSTANTE_INEXISTANTE');
```

La sortie obtenue en exécutant ces quelques lignes sera la suivante :

```
Constante MONEXT_CTE_STR_1 : Constante depuis extension
Constante PHP_CTE_DOUBLE_1 : 3.1415
```

La fonction `zend_get_constant()` est capable de lire aussi bien les valeurs de constantes définies depuis PHP que depuis une extension, et n'affiche pas d'avertissement en cas de constante inexistante. Elle retourne `1` en cas de succès et `0` en cas d'échec.

PHP fournit aussi la fonction `zend_get_constant_ex()` qui va un peu plus loin, en permettant de spécifier la classe à laquelle une constante doit appartenir, ainsi qu'une série d'options.

2.1.5 Constantes namespacées

Pour déclarer des constantes dans un espace de noms, PHP fournit les macros `REGISTER_NS_*_CONSTANT()`.

Ces macros font appel aux mêmes fonctions que `REGISTER_*_CONSTANT()`, à la différence près que le nom de la constante est calculé à l'aide de la macro `ZEND_NS_NAME()` – qui concatène le nom de l'espace de noms et le nom de la constante, en intercalant entre les deux le séparateur d'espaces de noms.

2.2 Variables super-globales

PHP inclut un mécanisme de variables appelées *super-globales*, qui sont accessibles depuis n'importe quelle fonction, sans avoir à être déclarées manuellement comme globales à l'aide du mot-clef `global`. Ces variables, traditionnellement, ont un nom en majuscules, qui commence par un underscore, comme `$_POST`, `$_GET`, ou `$_FILES`.

Une super-globale doit être connue du moteur de PHP avant que celui-ci ne commence à exécuter un script. En conséquence, il n'est pas possible d'en déclarer depuis du code PHP utilisateur. Par contre, une extension peut tout à fait créer des super-globales – d'ailleurs, `$_SESSION` est définie par l'extension `ext/session` !

2.2.1 Déclarer une variable super-globale

Déclarer une variable super-globale se fait depuis la fonction de votre extension branchée sur la phase `MINIT` de PHP, en appelant `zend_register_auto_global()` :

```
PHP_MINIT_FUNCTION(monext)
{
    zend_register_auto_global("_MAVAR", sizeof("_MAVAR") - 1, 0, NULL TSRMLS_CC);

    return SUCCESS;
}
```

Cette fonction attend en paramètres :

- le nom de la variable super-globale,
- la longueur de ce nom – sans compter le caractère nul de fin de chaîne,
- un paramètre booléen indiquant si la variable doit être initialisée “juste à temps” (JIT : Just In Time) ; spécifions pour l’instant `0` et nous verrons un peu plus loin ce que ce paramètre permet lorsqu’on lui passe `1`,
- et finalement et optionnellement, une fonction qui sera appelée lors de la phase de compilation du script PHP, à chaque fois que le nom de la variable sera rencontré.

Cette déclaration aurait aussi pu s’écrire en appelant la macro `ZEND_STRL()`, plutôt que d’écrire manuellement les deux premiers paramètres :

```
zend_register_auto_global(ZEND_STRL("_MAVAR"), 0, NULL TSRMLS_CC);
```

Pour illustrer l’utilisation de cette variable super-globale, faisons appel au script `test.php` suivant :

```
<?php
ma_fonction_init();
ma_fonction_affiche();

function ma_fonction_init()
{
    // Initialisation de la variable,
    // sans la spécifier comme "global"
    $_MAVAR = "Bonjour, Monde !";
}

function ma_fonction_affiche()
{
    // Utilisation de la variable,
    // toujours sans la noter "global"
    var_dump($_MAVAR);
}
?>
```

Ce script utilise la variable `$_MAVAR` depuis deux fonctions, une fois en écriture et une fois en lecture, sans jamais utiliser le mot-clef `global` ni le tableau `$GLOBALS`. Ceci ne fonctionne qu'avec une variable super-globale et ne serait pas possible avec une variable normale, chaque fonction ayant son espace de variables propre.

Exécuter cette portion de code donnera la sortie reproduite ci-dessous :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
string(16) "Bonjour, Monde !"
```

La première fonction a renseigné une valeur dans notre variable, et la seconde a pu la lire – nous avons donc bien créé une variable super-globale !

Notez que l'appel de `zend_register_auto_global()` permet de déclarer la variable, mais pas de l'initialiser. Cela signifie que si nous y accédons, depuis du code PHP, en lecture avant d'avoir écrit une valeur dedans, nous obtiendrons une notice !

La variable devant généralement avoir une valeur différente pour chaque requête, son initialisation peut être faite depuis la fonction branchée sur la phase `RINIT` de PHP – ou alors, comme nous allons le voir un peu plus bas.

2.2.2 Détecter les utilisations d'une super-globale

Plus haut, nous avons passé `NULL` en quatrième paramètre à la fonction `zend_register_auto_global()`. Celle-ci attend, pour ce quatrième paramètre optionnel, une fonction qui sera appelée lors de la compilation des scripts PHP, à chaque fois que la variable super-globale sera rencontrée.

En reprenant le code écrit précédemment, nous pouvons le modifier de la manière suivante :

```
PHP_MINIT_FUNCTION(monext)
{
    zend_register_auto_global("_MAVAR", sizeof("_MAVAR") - 1, 0,
        php_monext_auto_globals_create_mavar TSRMLS_CC);

    return SUCCESS;
}
```

Déclarons ensuite la fonction correspondante : elle reçoit en paramètres le nom de la variable super-globale et la longueur de ce nom, et retourne un booléen.

Ici, nous utilisons cette fonction pour initialiser notre variable `$_MAVAR`, en lui faisant correspondre un tableau associatif qui contiendra un élément :

```
static zend_bool php_monext_auto_globals_create_mavar(const char *name,
    uint name_len TSRMLS_DC)
{
    zval *valeur;

    php_printf("CREATE_MAVAR\n");

    ALLOC_ZVAL(valeur);
    array_init(valeur);
    INIT_PZVAL(valeur);

    add_assoc_string(valeur, "plop", "Bonjour", 1);

    zend_hash_update(&EG(symbol_table), "_MAVAR", sizeof("_MAVAR"),
        &valeur, sizeof(zval *), NULL);

    return 0;
}
```

Si cette fonction retourne `1`, alors, elle sera à nouveau appelée si la variable super-globale est à nouveau rencontrée plus loin dans le script PHP, lors de sa phase de compilation. Par contre, si cette fonction a retourné `0`, alors, elle ne sera plus appelée.

Pour tester cette modification apportée à notre extension, utilisons le script `test.php` suivant :

```
<?php
ma_fonction();

function ma_fonction()
{
    var_dump($_MAVAR);
}
?>
```

Puisque, désormais, notre extension initialise la variable `$_MAVAR`, exécuter ce script PHP donnera la sortie suivante :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
array(1) {
  ["plop"]=>
  string(7) "Bonjour"
}
```

Autrement dit, nous pouvons à présent, depuis du code utilisateur, accéder à la variable super-globale en lecture sans avoir auparavant écrit dedans : ceci est maintenant fait directement depuis l'extension !

Au sein du moteur de PHP, c'est typiquement de cette manière que sont renseignées les variables super-globales `$_GET` et `$_POST`.

2.2.3 Et avec JIT ?

Jusqu'à présent, lorsque nous avons appelé `zend_register_auto_global()` en lui passant en paramètre un nom de fonction à appeler lorsque la variable serait rencontrée lors de la phase de compilation d'un script PHP, nous avons toujours passé la valeur `0` en troisième paramètre.

Ce troisième paramètre est nommé `jit` (JIT, Just In Time – Juste à Temps). Si nous passons `0` comme nous l'avons fait à présent :

- la fonction sera appelée une première fois,
- puis elle sera appelée à chaque fois que la variable sera rencontrée lors de la compilation du script PHP, tant que la fonction n'a pas retourné `0` pour désactiver ces appels.

Dans le cas où la variable globale n'est pas du tout utilisée dans le script PHP exécuté, la fonction sera jouée une première fois, typiquement pour initialiser la variable – alors qu'il y a de fortes chances que cela soit complètement inutile (puisque la variable en question ne figure même pas dans le script).

Par exemple, modifions un peu notre extension et notre script PHP pour :

- que l'extension affiche un message depuis la fonction branchée sur la phase `MINIT` et depuis la fonction qui initialise la variable super-globale (la fonction de rappel branchée en quatrième paramètre lors de la déclaration de celle-ci),
- et que le script PHP affiche `AVANT` et `APRES` autour de l'appel de `ma_fonction()`

Exécuter ce script avec l'extension affichant plus d'informations de débogage donnera la sortie suivante :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
MINIT
CREATE_MAVAR
AVANT
array(1) {
  ["plop"]=>
  string(7) "Bonjour"
}
APRES
```

Ici, la variable super-globale est utilisée dans le script PHP et le paramètre `jit` est passé à `0` comme précédemment. La fonction utilisée pour initialiser la super-globale est donc appelée.

Si la variable était présente dans le code PHP, mais sans être utilisée (par exemple, si elle figurait dans le corps d'une fonction non invoquée), le comportement serait exactement le même : la super-globale est détectée à la compilation du code PHP, et non à son exécution.

Par contre, si le code de l'extension est modifié pour que le paramètre `jit` soit passé à `1`, alors, la fonction de rappel ne sera appelée que si la variable super-globale figure effectivement dans le code du script PHP.

Autrement dit, si le nom de la variable `$_MAVAR` ne figure pas du tout dans le script PHP, la sortie obtenue sera la suivante :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
MINIT
AVANT
APRES
```

Et si la variable est présente dans le code PHP au moins une fois, nous retrouvons la même sortie que précédemment, lorsque le JIT était désactivé :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
MINIT
CREATE_MAVAR
AVANT
array(1) {
  ["plop"]=>
  string(7) "Bonjour"
}
APRES
```

Et, pour insister un peu sur ce que je disais plus haut, si la variable est présente dans le code PHP, mais sans être utilisée, nous obtenons ceci, qui prouve que la fonction de rappel définie dans l'extension est bien appelée :

```
$ $HOME/bin/php-5.4-debug/bin/php -dextension=modules/monext.so -f ./test.php
MINIT
CREATE_MAVAR
AVANT
APRES
```

Ce troisième paramètre `jit` accepté par la fonction `zend_register_auto_global()` permet d'éviter une initialisation, potentiellement coûteuse, de variables super-globales, dans le cas où elles ne sont absolument pas utilisées dans un script PHP.

Il est utilisé au niveau du moteur de PHP qui spécifie une valeur de `1` pour la variable `$GLOBALS` (qui n'est que rarement utilisée – son initialisation n'est donc souvent pas utile), valeur qui peut aussi être spécifiée pour `$_SERVER`, `$_ENV`, et `$_REQUEST` en fonction de la valeur de la directive de configuration `auto_globals_jit`[4] (la première étant coûteuse à construire, et les deux suivantes peu utilisées).

2.3 Personnaliser la sortie de `phpinfo()`

La section de votre extension sur la page générée par la fonction `phpinfo()`, ou via `php -i` en ligne de commandes, est à ne pas négliger : elle sera vue par une bonne partie des développeurs utilisant les fonctionnalités de votre extension, mais aussi probablement par ceux qui devront installer, paramétrer ou administrer leurs applications.

Elle doit donc reprendre les informations principales de votre extension ; à savoir, en général :

- le nom de l'extension, pour indiquer qu'elle est chargée correctement,
- sa version,
- ses paramètres de configuration `.ini`,
- et, si l'extension dépend d'une bibliothèque externe, de la version de cette bibliothèque contre laquelle l'extension a été compilée.

Par défaut, si vous n'effectuez aucun développement spécifique à ce niveau, la sortie de `phpinfo()` reprendra les trois premiers points ; par exemple, pour une extension n'exposant aucune directive de configuration `.ini` :

[4] http://fr2.php.net/manual/fr/ini.core.php#ini.auto-globals-jit

Affichage de `phpinfo()`

Mais il est possible de personnaliser cette sortie, pour ajouter ou ne pas faire figurer certaines informations.

2.3.1 Personnaliser `phpinfo()`

Pour déclarer au moteur de PHP que votre extension contient une fonction qui se chargera de générer la section de `phpinfo()` lui correspondant, il faut ajouter une référence à celle-ci dans la structure `zend_module_entry` de l'extension, à l'aide de la macro `PHP_MINFO()` :

```
zend_module_entry monext_module_entry = {
    STANDARD_MODULE_HEADER,
    "monext",
    NULL, /* Function entries */
    NULL, /* Module init */
    NULL, /* Module shutdown */
    NULL, /* Request init */
    NULL, /* Request shutdown */
    PHP_MINFO(monext), /* Module information */
    "0.1", /* Replace with version number for your extension */
    STANDARD_MODULE_PROPERTIES
};
```

Le prototype de la fonction correspondante est ajouté dans le fichier `.h` de notre extension :

```
PHP_MINFO_FUNCTION(monext);
```

Et sa définition trouve sa place dans le fichier `.c` correspondant :

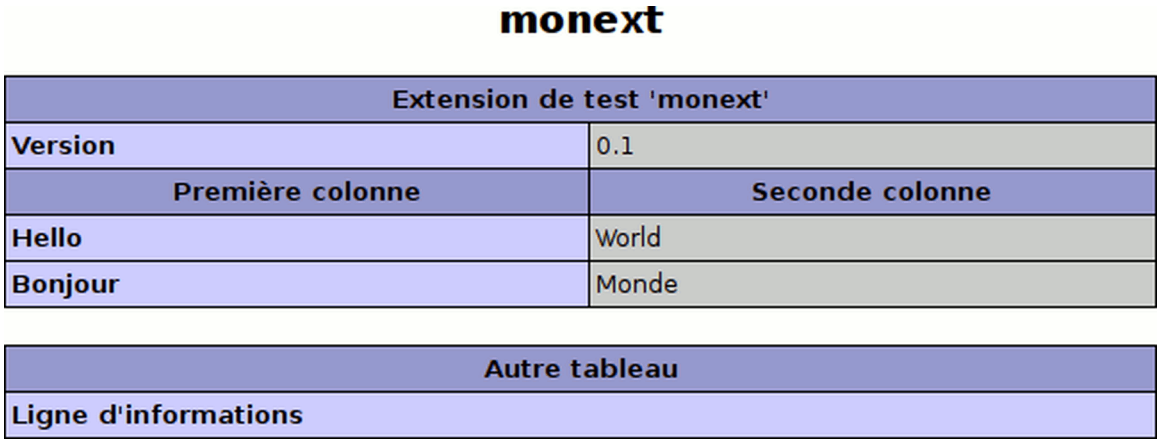
```
PHP_MINFO_FUNCTION(monext)
{
    php_info_print_table_start();

    php_info_print_table_colspan_header(2, "Extension de test 'monext'");
    php_info_print_table_row(2, "Version", "0.1");

    php_info_print_table_header(2, "Première colonne", "Seconde colonne");
    php_info_print_table_row(2, "Hello", "World");
    php_info_print_table_row(2, "Bonjour", "Monde");

    php_info_print_table_end();


    php_info_print_table_start();
    php_info_print_table_header(1, "Autre tableau");
    php_info_print_table_row(1, "Ligne d'informations");
    php_info_print_table_end();
}
```

Les sorties correspondant à `phpinfo()` sont généralement générées par le biais de fonctions dont le nom est de la forme `php_info_print_*()`.

Ici, la sortie générée se compose d'un premier tableau, délimité par les appels à `php_info_print_table_start()` et `php_info_print_table_end()`, à deux colonnes :

- la première ligne est une ligne de titres, contenant une seule colonne occupant la largeur de deux, via un attribut HTML `colspan` que l'on retrouve dans le nom de la fonction `php_info_print_table_colspan_header()`,
- la seconde ligne est composée de deux colonnes, affichant la version de notre extension,
- vient ensuite une nouvelle ligne de titre, de deux colonnes cette fois-ci,
- Suivie de deux lignes de deux colonnes contenant quelques mots.

La sortie se poursuit ensuite avec un second tableau, et le rendu en sortie HTML serait le suivant :



monext

Extension de test 'monext'	
Version	0.1
Première colonne	Seconde colonne
Hello	World
Bonjour	Monde

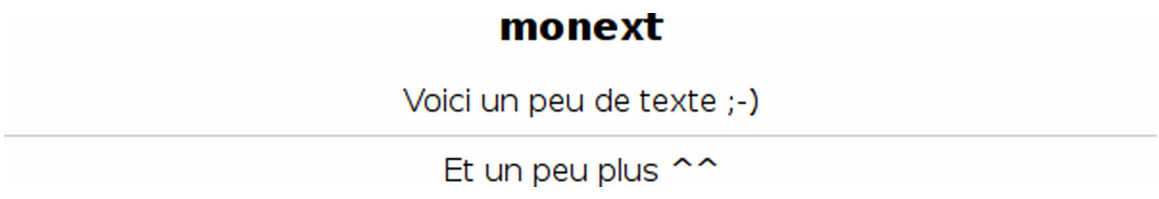
Autre tableau
Ligne d'informations

`phpinfo()` **personnalisé : affichage de tableaux**

Il est cela dit possible de ne pas utiliser ces fonctions de tableaux et d'écrire du texte sur la sortie standard, comme le fait la fonction ci-dessous, qui sépare deux lignes de texte par une ligne horizontale (balise HTML `<hr />`) :

```
PHP_MINFO_FUNCTION(monext)
{
    /* Texte en dehors de tout conteneur */
    php_printf("Voici un peu de texte ;-)");

    /* <hr /> ; ou 31 "_" si sortie non HTML */
    php_info_print_hr();

    /* Texte en dehors de tout conteneur (suite) */
    php_printf("Et un peu plus ^^");
}
```

La sortie HTML ressemblerait alors à la capture d'écran reproduite ci-dessous :



`phpinfo()` **personnalisé : affichage sur la sortie standard**

Cette sortie n'est pas vraiment *jolie* et ne met pas vraiment notre extension en valeur… À éviter, donc !

En sortie texte, si le script PHP contenant l'appel à `phpinfo()` est invoqué en ligne de commandes, ou via `php -l`, nous obtiendrions la sortie suivante :

```
monext
Voici un peu de texte ;-)

_____________________________________________________________________

Et un peu plus ^^
```

Notez que la ligne horizontale a automatiquement été transformée par PHP en 71 caractères "_", correspondant à une *ligne*.

Dans le cas où une sortie sous forme de tableaux ne vous conviendrait pas, et pour éviter cette sortie *brute* visuellement peu satisfaisante, il est possible de positionner des *boites* dans la sortie de `phpinfo()` : des zones de texte, dont les styles sont en accord avec le reste de la sortie, où les affichages sont libres.

La création d'une telle zone est entourée d'un appel aux fonctions `php_info_print_box_start()` et `php_info_print_box_end()`, la première acceptant `0` ou `1` en paramètre, en fonction du style souhaité pour la boite. La sortie textuelle entre ces deux appels se fait vers la sortie standard, et vous êtes libre d'y positionner ce que vous souhaitez. Par exemple :

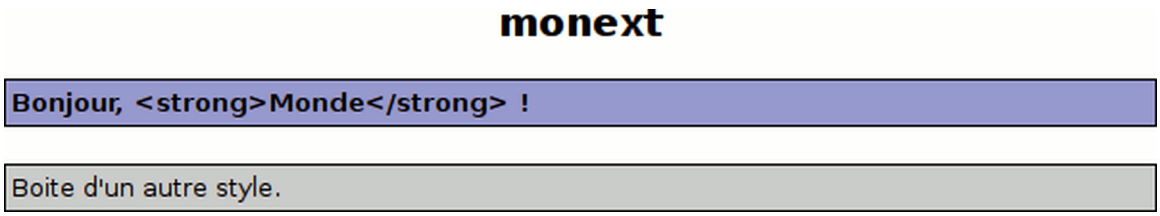
```
PHP_MINFO_FUNCTION(monext)
{
    char html[] = "Bonjour, <strong>Monde</strong> !";
    char *html_escaped = php_info_html_esc(html TSRMLS_CC);

    php_info_print_box_start(1);

    php_printf("%s", html_escaped);

    /*
      php_info_html_esc() retourne une chaîne dont l'espace a été alloué
      en mémoire => il faut le libérer nous-mêmes
    */
    efree(html_escaped);

    php_info_print_box_end();

    php_info_print_box_start(0);
    php_printf("Boite d'un autre style.");
    php_info_print_box_end();
}
```

Notez que nous avons ici utilisé la fonction `php_info_html_esc()` pour encoder une chaîne de caractères à afficher, qui contenait des balises HTML que nous ne souhaitions pas voir interprétées ;

cette fonction peut être considérée comme équivalente à la fonction utilisateur `html_entities()`, adaptée à une sortie de `phpinfo()`.

La fonction `php_info_html_esc()` retourne une nouvelle chaîne de caractères, allouée en mémoire, qu'il vous revient donc de libérer lorsque vous n'en avez plus besoin !

La sortie HTML obtenue avec cette portion de code ressemblerait à ceci :



`phpinfo()` personnalisé : boites et sortie HTML échappée

Vous pouvez remarquer les deux styles différents pour chacune des deux zones de texte et le fait que les balises HTML présentes dans le texte de la première zone ont bien été échappées et ne sont donc pas interprétées.

En interne, le moteur de PHP invoquera la fonction `php_info_print_module()`. C'est elle qui affiche le nom du module et sa version si vous ne définissez pas de fonction chargée de la génération de la section de `phpinfo()` correspondant à votre extension, ou appelle la fonction `MINFO()` si vous en avez défini une.

2.3.2 Sortie HTML / sortie textuelle

La fonction `phpinfo()` n'est pas toujours appelée pour générer une sortie HTML : en fonction de la SAPI, il se peut qu'une sortie textuelle doive être générée – typiquement, lorsque `phpinfo()` est appelée depuis un script exécuté en ligne de commande, ou directement via `php -i`.

Si la sortie que vous cherchez à générer est susceptible de contenir du HTML, il vous faudra donc veiller à prévoir une version textuelle, qui restera lisible lorsqu'une sortie plus *brute* sera générée.

Le fichier `SAPI.h` contient ce qu'il nous faut pour déterminer si la SAPI courante correspond à une génération de `phpinfo()` en mode textuel ; incluons donc ce fichier d'en-têtes depuis le fichier `.h` de notre extension :

```
#include "SAPI.h"
```

Et ensuite, dans la fonction `MINFO()` de notre extension, nous pouvons accéder à `sapi_module.phpinfo_as_text`, qui sera vraie s'il nous faut générer une sortie textuelle. Par exemple, nous pourrions envisager l'utilisation d'un portion de code ressemblant à celle-ci :

```
PHP_MINFO_FUNCTION(monext)
{
    php_info_print_table_start();

    if (sapi_module.phpinfo_as_text)
    {
        php_info_print_table_row(2, "Texte", "Bonjour, **Monde** !");
    }
    else
    {
        /* Echappement HTML déjà fait en interne !!! */
        php_info_print_table_row(2, "Texte",
            "Bonjour, <strong>Monde</strong> !");
    }

    php_info_print_table_end();
}
```

La sortie obtenue en mode HTML serait la suivante :

`phpinfo()` personnalisé : sortie HTML

Notez que la fonction `php_info_print_table_row()` a elle-même pris le soin d'échapper les balises HTML que nous avions positionnées dans la chaîne de caractères affichée ! La condition utilisée ici, basée sur `sapi_module.phpinfo_as_text`, aurait donc plutôt tendance à être utile lorsque la sortie est affichée sans passer par les fonctions `php_info_*()`, comme lorsque nous produisons un affichage au sein d'un *boite*.

En mode textuel, c'est la première chaîne qui remonterait :

```
monext

Texte => Bonjour, **Monde** !
```

Pour rebondir sur ce que je disais quelques lignes plus haut, voici un exemple de code où nous générons une sortie dans une *boite* :

```
PHP_MINFO_FUNCTION(monext)
{
    php_info_print_box_start(0);
    if (sapi_module.phpinfo_as_text)
    {
        php_printf("Bonjour, **Monde** !");
    }
    else
    {
        php_printf("Bonjour, <strong>Monde</strong> !");
    }
    php_info_print_box_end();
}
```

Ici, la sortie HTML correspond à ce que nous attendions :

`phpinfo()` **personnalisé : boite et sortie HTML**

Veillez à ce que les sorties textuelles et HTML présentent les mêmes informations, de façon à ne pas *perdre* vos utilisateurs.

2.3.3 Affichage des directives `.ini`

À partir du moment où votre extension définit des directives de configuration `.ini`, il est recommandé de les faire remonter dans la sortie de `phpinfo()`. C'est même le fonctionnement par défaut si vous ne définissez pas de fonction `MINFO()`.

Pour plus d'informations, notamment sur la personnalisation de l'affichage des valeurs pour chaque directive, consultez la section `phpinfo()` du Chapitre « Configuration par fichier `.ini` »

3. Et maintenant ? Un livre complet !

Arrivés ici, vous avez parcouru l'introduction de mon livre **Développer une Extension PHP**, ainsi que l'intégralité de son premier chapitre et quelques sections qui figurent plus loin dans le livre.

Si ces quelques dizaines de pages ont – ce que j'espère – éveillé votre curiosité et que vous souhaitez en apprendre plus, vous pouvez acheter ce livre en version complète à l'adresse suivante :

> https ://leanpub.com/developper-une-extension-php[1]

Il est disponible aux formats électroniques PDF, EPUB et MOBI. Les éventuelles mises à jour, qui pourraient apporter quelques corrections et ajout de précisions ici et là sont bien entendu téléchargeables sans surcoût.

> Encore une fois, *je vous souhaite une excellente lecture et bon courage pour vos premiers pas dans le développement d'extensions PHP !*
>
> – **Pascal MARTIN**

[1] https://leanpub.com/developper-une-extension-php