PHP: Anatomía del mal diseño

Compilación de la web y adaptación de Jorge Amado Soria Ramírez. (05/12/2012)

El siguiente documento es una recopilación de varias fuentes de Internet, con una mezcla de opiniones personales, juicios subjetivos y razones técnicas. Es un poco más largo de lo que sería deseable para un artículo de divulgación científica, y entiendo que esto atente contra su mismo propósito. (En las palabras de Winston Churchill: “Este documento, por su propio tamaño, se defiende a sí mismo del riesgo de ser leído”). Por favor tengan tanta paciencia como sea posible con el contenido aquí recopilado.

Versiones

Fecha	Actividad	Autor
07/12/2012	Creación y publicación del documento	Jorge A. Soria
09/12/2012	Modificación de la sección sobre allow_url_fopen	Jorge A. Soria
11/12/2012	Modificación de la sección sobre la configuración con varios php.ini	Jorge A. Soria
04/06/2013	Adicionada sección sobre anotaciones	Jorge A. Soria

Introducción

Soy un gruñón. Me quejo de muchas cosas. Hay mucho en el mundo de la tecnología que no me gusta, y es normal que así sea – la programación es una disciplina muy joven, y ninguno de nosotros tiene ni la menor idea de lo que estamos haciendo. Se combina esto con la ley de Sturgeon (el noventa por ciento de todo es basura) y tenemos material para quejarnos toda la vida.

Pero este no es el caso. PHP no solamente es extraño de utilizar, o no es adecuado para lo que quiero, o es subóptimo, o contra mi religión. Puedo contar muchas cosas buenas de lenguajes que evito, y muchas cosas malas de lenguajes que me gustan. PHP es la única excepción. Virtualmente cada funcionalidad de PHP está rota de alguna manera. El lenguaje, el marco de trabajo, el ecosistema, todos son simplemente *malos*. Y no se puede aislar ninguna maldita cosa que esté dañada, porque el daño es sistémico, por todas partes. Cada vez que trato de reunir una lista de problemas de PHP, me quedo trabado en una búsqueda en profundidad, descubriendo más y más pequeñas dificultades triviales.

PHP es una *vergüenza*, una plaga sobre mi arte. Está tan roto y es tan celebrado por tantos aficionados que todavía no han aprendido ninguna otra cosa que resulta enloquecedor. Tiene muy escasas cualidades que lo redimen y en general, preferiría olvidar que existe en absoluto.

Analogía

No puedo siquiera decir exactamente que está mal con PHP, porque… bueno, a ver. Imaginemos que tenemos, no sé, una caja de herramientas. Un conjunto de herramientas normal, cosas estándares.

Sacas un destornillador, y ves que es uno de esos raros, con la cabeza triangular. Bueno, esto no es muy útil para ti, pero a lo mejor puede servir en algunas ocasiones.

Sacas el martillo, pero para tu desdicha, ves que tiene uñas en **los dos lados**. De todas maneras todavía es útil, quiero decir, siempre se puede clavar con el centro de la cabeza si lo sostienes de lado.

Sacas las pinzas, pero ves que no son serradas, sino lisas. Eso es menos útil, pero de todas maneras, con cierto esfuerzo, aún se pueden apretar tuercas con ellas, así que, como sea.

Y los carpinteros te enseñan las casas que han construido, donde cada cuarto es un pentágono y el techo esta al revés. Y cuando tocas a la puerta esta se cae, y los carpinteros te gritan por romperla.

Y así. Todo en la caja es un poco raro y particular, pero no lo suficiente para que sea **completamente** inútil. Y no hay un problema claro con el conjunto, todas las herramientas están ahí.

Ahora imagínense que conocen a millones de carpinteros que usan esta caja, y te dicen "bueno, y ¿cuál es el problema con estas herramientas? ¡Son las únicas que he usado y funcionan perfectamente!". Y los carpinteros te enseñan las casas que han construido, donde cada cuarto es un pentágono y el techo esta al revés. Y cuando tocas a la puerta esta se cae, y los carpinteros te gritan por romperla.

Eso es lo que está mal con PHP.

Posición

Sostengo que las siguientes cualidades son importantes para hacer de un lenguaje una herramienta productiva y útil. PHP las viola con total abandono. Si no estamos de acuerdo en que estas cualidades son cruciales, pues, no me imagino como vamos a ponernos de acuerdo sobre nada en absoluto.

*Un lenguaje debe ser **predecible**. Es un medio para expresar ideas humanas y para hacer que una computadora las ejecute, así que es crítico que la comprensión humana de un programa sea correcta.*

- Un lenguaje debe ser **predecible**. Es un medio para expresar ideas humanas y para hacer que una computadora las ejecute, así que es crítico que la comprensión humana de un programa sea correcta.
- Un lenguaje debe ser **consistente**. Las cosas similares deben verse similares, las diferentes deben verse diferentes. Conocer una parte del lenguaje debería ayudar a conocer el resto.
- Un lenguaje debe ser **conciso**. Los nuevos lenguajes surgen para reducir el boilerplate de viejos lenguajes (después de todo, cualquiera puede escribir código máquina). Un lenguaje debe por tanto tratar de evitar la introducción de código boilerplate propio.
- Un lenguaje debe ser **confiable**. Los lenguajes son herramientas para resolver problemas, deberían minimizar los problemas que introducen. Cualquier pequeña idiosincrasia del mismo constituye una distracción inútil y enorme.
- Un lenguaje debe ser **depurable**. Cuando algo sale mal, el programador debe arreglarlo, y necesita toda la ayuda posible para hacerlo.

Mi posición es la siguiente:

- PHP está lleno de sorpresas: `mysql_real_escape_string`, `E_ALL`
- PHP es inconsistente: `strpos`, `str_rot13`
- PHP requiere boilerplate: manejo de errores para el API C, `===`
- PHP es poco confiable: `==`, `foreach ($foo as &$bar)`
- PHP es opaco: no tiene stack traces por defecto ni para errores fatales, reporte de errores complejo

Muy importante, no argumento que PHP no sea una herramienta que pueda usarse para hacer grandes cosas. Eso sería fanatismo y además es una posición refutada por la realidad. Lo que se argumentará con ejemplos en este documento, es la incapacidad de PHP para fungir como una herramienta de desarrollo válida para aplicaciones robustas; la poca seriedad y madurez de su entorno y sus desarrolladores promedio; la dificultad del mantenimiento y extensión de las aplicaciones desarrolladas con PHP y finalmente, la falsedad de los argumentos que mantienen que PHP es una tecnología viable y eficiente para desarrollos empresariales o académicos sin una base anterior ya existente.

En otras palabras, no digo que no haya condiciones en las cuales PHP no sea la opción más adecuada. Sí digo que las situaciones en que estén justificadas estas decisiones nunca tienen una base tecnológica, sino que están condicionadas a razones económicas, personales, o administrativas que no pueden ser defendidas desde una posición técnica responsable.

Problema

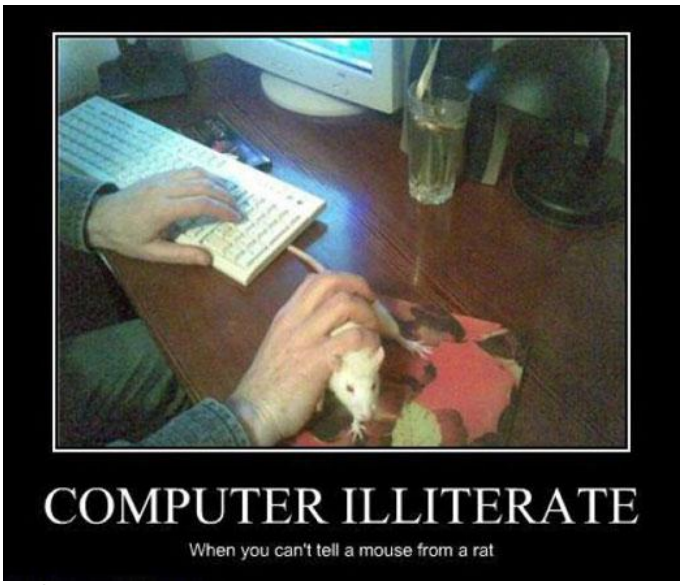
Tengo que decir esto aquí, porque vale la pena repetirlo: PHP es una comunidad de aficionados. Muy pocas personas que lo diseñan, que trabajan en él, o que escriben código usándolo parecen saber lo que están haciendo. Aquellos que **sí** saben lo que están haciendo tienden a moverse hacia otras plataformas, reduciendo la competencia promedio total del conjunto. Es un caso de ciegos conduciendo ciegos.

PHP es la herramienta adecuada para algunos trabajos. El problema es que es una herramienta fea, tosca y torpe que me hace llorar y tener pesadillas.

Esto es, claramente, una generalización. Por supuesto que existen buenos desarrolladores que trabajan sobre PHP, algunos incluso puede que por su propia voluntad. Pero como los carpinteros de la analogía, las probabilidades de que los desarrolladores que trabajan con PHP sean principiantes que no conocen otra herramienta son abrumadoras. Y es lógico, además. Veamos por qué.

Una persona que quiere hacer un sitio web y tiene conocimientos suficientes de programación para hacerlo, puede ver en PHP una panacea. O sea, tenemos un lenguaje fácil de usar, fácil de aprender, con muchas funciones para todas las tareas más básicas necesarias para hacer una aplicación web. Es la opción lógica… para **construir** la aplicación.

Pero, en el software, codificar es una parte necesaria, pero insuficiente, del desarrollo. Más aún a medida que el producto crece en clientes y código. El desarrollo de software es una actividad multifacética, compleja, que requiere del esfuerzo de muchas personas o en su defecto, de una persona con muchas habilidades. Y a medida que aumenta el tamaño (y la importancia) de las aplicaciones, la importancia de codificar disminuye dramáticamente en favor de otras actividades que también realizan los desarrolladores: depuración, pruebas, atención al cliente, documentación. En otras palabras, mientras mayor y más crítico el proyecto, menos importante es la capacidad del lenguaje de ser sencillo de codificar, y más importante es su capacidad de ser predecible, robusto y tener un buen soporte para pruebas y despliegue (se podría decir que PHP es fácil de desplegar, pero más adelante veremos que esto es una opinión subjetiva).



Aquí es donde se ve el problema con PHP. Es tan sencillo de aprender que codificar está al alcance de muchas personas. Personas que no están listas para desarrollar aplicaciones reales aún, sin la disciplina necesaria para diseñar, probar, desplegar, documentar y soportar sus aplicaciones.

Aquí es donde se ve el problema con PHP. Es tan sencillo de aprender que codificar está al alcance de muchas personas. Personas que no están listas para desarrollar aplicaciones reales aún, sin la disciplina necesaria para diseñar, probar, desplegar, documentar y soportar sus aplicaciones. Así que, estas personas desarrollan sistemas, que con el tiempo crecen. La mayoría muere por su propio peso, pero los que sobreviven, aquellos hechos por la mayor cantidad de personas o soportados por una comunidad numerosa, o hechos por personas realmente talentosas, esos crecen y son adoptados y mantenidos por una gran cantidad de personas, y utilizados por estas (con intenciones maliciosas o no) para justificar el uso de este ecosistema tóxico y primitivo como una opción realmente válida en el contexto del desarrollo empresarial.

Se llega al punto de que muchos programadores ni siquiera tienen una opción al respecto, porque la opción viene decidida desde arriba, por administradores que imponen la selección basada en criterios que no tienen nada que ver con los méritos técnicos. Y lo peor es, que a veces funciona. Después de todo, la calidad de un sistema no es garantía de su éxito, es tan solo una gran ayuda.

Yo no sé ustedes, pero para mí construir software es lo suficientemente difícil como para considerar que necesitamos toda la ayuda posible. Empezando por nuestras herramientas.

Para ser honestos, reconozcamos desde el principio las cosas buenas de PHP. Vamos a hacerlo para que después los lectores no piensen que no las conocemos y venga con comentarios del tipo "tienes razón, pero PHP hace esto o aquello".

- PHP hace que los principiantes lo tengan fácil

  Es correcto, y como vimos antes, estoy de acuerdo. Yo he pasado cursos de PHP, incluso fue mi primer lenguaje Web. Mientras todos los cursos introductorios de Java pasan mucho tiempo explicando cosas que son difíciles de entender para los novatos, como OOP y semejantes, PHP te deja conectarte a una base de datos inmediatamente. El soporte de MySQL es directo y sencillo. Una página PHP al principio corresponde exactamente a una página HTML. Es el tipo de concepto que te hace sentir poderoso. Lo cual es algo que la mayoría de los demás lenguajes no hacen.

  PHP te lo pone fácil al principio, más que nada. Es lo que a las personas que ya tienen conocimiento de OOP les gusta de Ruby on Rails.

- PHP es fácil de instalar

Es cierto. De hecho existen múltiples paquetes, como el XAMPP, que permiten comenzar a trabajar instantáneamente. Sin embargo, una vez que se selecciona un lenguaje de programación y sistema de plantillas, la instalación no debería demorar más de una hora, incluyendo casos peores.

Esta es una operación que se hace una sola vez. Si tiene que hacerse en varios cientos de máquinas, puede automatizarse. Cuando cientos o miles de horas de esfuerzo se gastan en codificar, ¿es realmente tan importante el tiempo que gastas en instalar un lenguaje de programación y un sistema de plantillas?

- PHP es más fácil de aprender que Perl/Python/Ruby/etc.

  Aprender lo básico de cualquier lenguaje no es difícil. Aprender buenas prácticas que promuevan mantenibilidad, seguridad, simplicidad, robustez, etc., requiere tiempo y experiencia. La mayor parte de los codificadores que tienen este tipo de experiencia no tienen ningún problema aprendiendo otro lenguaje. Los que no, escribirán código malo sin importar el lenguaje.

  La documentación de PHP es incompleta. Los comentarios en el sitio de la ayuda son muchas veces incorrectos. Las traducciones son incompletas y la información provista no es consistente (incluye datos sobre otros lenguajes o herramientas que no dependen de PHP directamente, ej. MongoDB). Se puede decir que esto en realidad es un obstáculo para aprender a programar PHP apropiadamente.

- Más personas conocen PHP que Perl/Python/Ruby/etc.

  Quizás sea cierto, pero ¿acaso más buenos programadores conocen PHP que Perl/Python/Ruby/etc.? Si no pueden elegir otros lenguajes, ¿realmente vale la pena trabajar con ellos?

- ¡PHP facilita la construcción de sitios Web! ¡PHP provee sistema de plantillas y código embebido en el HTML por defecto!

  Dejando a un lado por un momento el argumento de que esto no es una buena funcionalidad para construir sistemas grandes, debido a que alienta la mezcla de lógica de negocio y código de presentación (aunque se pueden adquirir hábitos para evitarlo), estos sistemas de plantilla y código embebido en HTML existen para otros lenguajes. La única ventaja de PHP es que lo provee por defecto como funcionalidad básica.

Desafortunadamente, aquí termina la diversión. Por esto es que PHP es un juguete, no una herramienta, y no puedo repetir esto lo suficiente. Si están planeando desarrollar un sistema de software y se puede tomar una decisión en cuanto a la tecnología, no hay casi **ninguna** razón por la cual elegir a PHP, pero **sí** hay muchísimas para no elegirlo.

Razones

Bueno, pero aún el lector podría pensar: "Vale, no te gusta PHP, incluso tienes algunas razones para ello. Es tu decisión, y la mía es usarlo, ¿Cuál es el problema? ¿Qué te da derecho a decir que lo que hago está mal sin siquiera conocerme? ¡Mis decisiones no son de tu incumbencia!"

Pues, sí que lo son. Según Tim Bray:

"basándome en mi limitada experiencia (…) todo el código PHP que he visto ha sido basura desordenada e inmantenible. Espagueti SQL dentro de espagueti PHP dentro de espagueti HTML, replicado en formas ligeramente distintas en docenas de lugares."

Y es algo en lo que no puedo evitar estar de acuerdo, porque esa es también mi experiencia. Si yo debo lidiar con los resultados de las malas decisiones de los demás, eso convierte a esas decisiones en mi problema, ¿no les parece?

Pero la mayor razón por la cual me veo forzado a compilar este documento, es para impulsar la singularidad de PHP.

Basándome en mi limitada experiencia (…) todo el código PHP que he visto ha sido basura desordenada e inmantenible. Espagueti SQL dentro de espagueti PHP dentro de espagueti HTML, replicado en formas ligeramente distintas en docenas de lugares.

La singularidad

¿Apesta PHP? Por supuesto que sí. Es una supernova galáctica de incomprensible, colosal, enloquecedoramente horrible basura. Si alguna vez han programado en PHP y tienen al menos una onza de talento en su cuerpo, no es posible que hayan sacado otra conclusión. Es inevitable.

Pero aun así, eso **no importa**. Algunos de los mayores sitios de Internet (sitios que utilizamos todos los días) están escritos en PHP. Si PHP es tan profundamente malo, **¿Por qué se utiliza en tantos lugares de Internet?**

La conclusión que saco es que **construir una aplicación atractiva es mucho más importante que elegir el lenguaje**. Así que aunque PHP no sea mi elección, y si me preguntan tengo que decir que **nunca** debería ser la elección de ningún humano racional que se siente delante de una computadora, no se puede discutir con los resultados.

Probablemente hayan escuchado que codificadores suficientemente incompetentes pueden escribir FORTRAN en cualquier lenguaje. Bueno, pues el inverso también es cierto: **codificadores suficientemente talentosos pueden escribir grandes aplicaciones en lenguajes terribles**. Es una lección dolorosa, pero importante.

Lo más deprimente no es que PHP este diseñado horriblemente. ¿Acaso alguien disputa que PHP es de entre los lenguajes principales el peor diseñado, la mayor contaminación sobre nuestra práctica en décadas? Lo verdaderamente deprimente no es esto, sino que en años **no ha cambiado casi nada**.

¿Está PHP tan roto que no puede usarse? No, claramente no. El gran crimen de PHP es su completa banalidad. Su continua popularidad es la prueba viva de que la calidad es irrelevante: barato, popular y en todas partes **siempre gana**. PHP es el reggaetón de los lenguajes de programación.

La metáfora del martillo de dos uñas es muy apta, porque en esencia, el tema principal aquí es la selección de la herramienta correcta. Como notaba Alex Papadimoulis:

"Un cliente me ha pedido que construya e instale un sistema de estantería personalizado. Estoy en el punto en que necesito clavarlo, pero no estoy seguro de que usar para golpear los clavos. ¿Debería usar un zapato viejo o una botella de cristal?

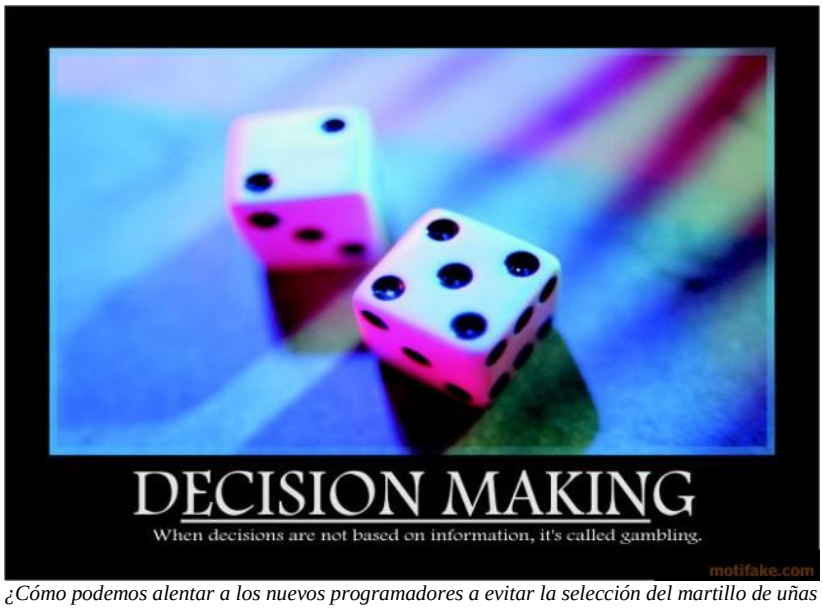
¿Cómo se responde esta pregunta?

Un cliente me ha pedido que construya e instale un sistema de estantería personalizado. Estoy en el punto en que necesito clavarlo, pero no estoy seguro de que usar para golpear los clavos. ¿Debería usar un zapato viejo o una botella de cristal?

a. *Depende. Si lo que se quiere es clavar un clavo pequeño en yeso, la botella es probablemente más sencilla de usar, especialmente si el zapato está sucio. Sin embargo, si lo necesario es clavar un clavo grueso en madera, entonces elige el zapato: la botella se romperá en tu mano.*

b. *Hay algo fundamentalmente mal con la manera en la que estás trabajando: necesitas herramientas reales. Sí, puede que sea necesario ir a la caja de herramientas, o incluso a la ferretería, pero hacerlo bien te ahorrará tiempo, dinero y agravios a todo lo largo de la vida de tu producto. Necesitas dejar de construir cosas por dinero hasta que entiendas los principios de la construcción."*

De lo que deberíamos estar hablando no es de lo terrible que es PHP, sino de cómo los programadores podemos desplazar culturalmente una herramienta tan profundamente dañada con otra mejor. **¿Cómo podemos alentar a los nuevos programadores a evitar la selección del martillo de uñas dobles para que elijan… cualquier otro martillo normal?**



¿Cómo podemos alentar a los nuevos programadores a evitar la selección del martillo de uñas dobles para que elijan… cualquier otro martillo normal?

Desde mi perspectiva, el punto de todos los artículos anti PHP no es solo quejarse, sino ayudar a educar y potencialmente advertir a los nuevos programadores que están comenzando nuevos proyectos. Varios trabajos muy buenos, históricos incluso, se han hecho con PHP a pesar de su incuestionable locura. Pero ahora necesitamos ponernos a trabajar todos juntos para arreglar lo que está roto. La mejor manera de arreglar el problema de PHP en este punto **es hacer que las alternativas sobresalgan tanto que la elección de cuál es el mejor martillo sea obvia.**

Ese momento será la singularidad de PHP que estoy esperando. Y estoy intentando lo mejor que puedo que esta singularidad ocurra.

La mejor manera de arreglar el problema de PHP en este punto ***es hacer que las alternativas sobresalgan tanto que la elección de cuál es el mejor martillo sea obvia.***

Es necesario evitar más proyectos de PHP para lograr este momento especial, esta singularidad. Una manera de propulsar el desuso de PHP es proporcionarle apoyo a esas personas que conocen, de una manera instintiva, que el lenguaje es una aberración creada en una noche sin luna en el fondo del infierno mismo, pero no pueden apuntar a una característica específica del lenguaje o las herramientas en una discusión. Este es mi intento de hacer mi parte, y quisiera que ustedes también hicieran la suya. Hey, al menos tengo que intentarlo.

No comenten con esto

He estado en **muchas** discusiones de PHP. He oído muchos contra argumentos genéricos que están diseñados tan solo para detener las conversaciones inmediatamente. No contesten con estas cosas, por favor:

C++ permite que te vueles tu propio pie con una escopeta si no tienes cuidado. PHP hace que sea posible que tu pie le dispare a la escopeta.

- No digan que "los buenos desarrolladores pueden escribir buen código en cualquier lenguaje", bla bla bla. Eso no significa nada. Un bueno carpintero puede clavar un clavo con una roca o con un martillo, pero ¿cuántos carpinteros ven golpeando cosas con piedras? Parte de ser un buen desarrollador es la habilidad de elegir las herramientas que funcionan mejor.
- No digan que es responsabilidad del desarrollador memorizar mil excepciones raras y comportamientos sorprendentes. Esto es necesario en todos los sistemas, porque las computadoras son ineficientes. Eso no significa que no haya límite para la cantidad de locura aceptable en un sistema. PHP no es nada sino excepciones, y no es bueno tener que luchar más con el lenguaje que con el programa que se está escribiendo. Mis herramientas no deben crear más trabajo neto que hacer. Parafraseando a Stroustrup, C++ permite que te vueles tu propio pie con una escopeta si no tienes cuidado. PHP hace que sea posible que tu pie le dispare a la escopeta.
- No digan que "así funciona el API de C". ¿Cuál es la idea de utilizar un lenguaje de alto nivel si todo lo que hace es ofrecer ayuda para el trabajo con cadenas y una tonelada de wrappers en C? ¡Simplemente programa en C! Incluso existe una librería CGI para ello.
- No digan "eso es lo que pasa por tratar de hacer cosas raras". Si dos funcionalidades existen, alguien va a tener una razón para usarlas juntas. Y de nuevo, esto no es C, no hay una especificación, no hay necesidad para tener "comportamiento indefinido".
- No digan que Facebook o la Wikipedia están hechas en PHP. ¡Lo sé! Podrían estar escritos en Brainfuck, que mientras haya suficientes personas listas para enfrentar las cosas, los problemas con una plataforma pueden ser superados. Hasta donde sabemos, el tiempo de desarrollo podría haberse reducido a la mitad o haberse duplicado si estos productos se hubieran escrito en otros lenguajes. Este dato por sí solo no significa nada.
- No digan que "PHP es más rápido, más escalable, más apropiado para la Web". La Web no son los sitios webs y las páginas ricas en texto e imágenes, es mucho más. Dentro de la Web hay muchos servicios, intensivos en computación, como redes de cálculo, VPNs, gráficos, etc., que no son el punto fuerte de PHP. Incluso dentro de los sitios web, Facebook tendrá un billón de visitas mensuales, pero Google tiene un billón de visitas mensuales **por usuarios únicos**. Y no está hecho en PHP. De nuevo, este dato por sí solo no significa nada.

- En realidad, no digan nada. Esta lista es mi oportunidad de decir lo que pienso. Si leyendo esto no cambia la idea que tienen de PHP, nada lo hará, así que en vez de opinar contra alguien en la red simplemente dedíquense a hacer un sitio web en tiempo récord y prueben que estoy equivocado. :)

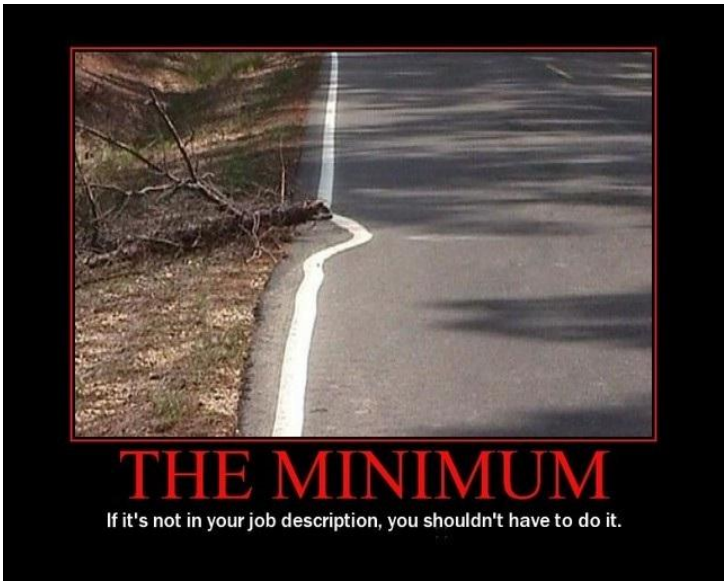
Entre paréntesis, Java me encanta. No pretendo decir que sea perfecto, simplemente mido los beneficios contra los problemas y veo que Java es la mejor opción para mi trabajo. Nunca he conocido a ningún desarrollador de PHP que pueda decir lo mismo. Me he encontrado con muchos que rápidamente comienzan a justificar y a disculparse por todas las cosas que PHP hace. Este estado mental es francamente aterrador.

PHP

Filosofía

- PHP fue originalmente diseñado para no-programadores (y leyendo entre líneas, no-programas). Nunca ha escapado completamente de estas raíces. Una cita selecta de la documentación de PHP 2.0 acerca de + y sus amigos haciendo conversión de tipos:

  "Una vez que se comienza a tener operadores separados para cada tipo se empieza a hacer el lenguaje mucho más complejo. Por ejemplo, no se puede usar == para cadenas, sino que tienes que usar `eq`*. No vemos por qué esto debería ser de esta forma, especialmente en PHP donde la mayor parte de los scripts son bastante simples y en la mayoría de los casos escritos por no programadores que quieren un lenguaje con una sintaxis lógica básica que no sea muy difícil de aprender."*



PHP fue originalmente diseñado para no-programadores (y leyendo entre líneas, no-programas).

- PHP está hecho para seguir funcionando a cualquier precio. Cuando se presenta la opción entre hacer algo sin sentido o abortar el programa con un error, PHP hará algo sin sentido. La filosofía de "algo es mejor que nada".
- No hay una filosofía de diseño clara. El PHP temprano estaba inspirado por Perl; la gran `stdlib` con parámetros de salida es de C; las partes de OOP están diseñadas como C++ y Java.
- PHP toma vastas cantidades de inspiración de otros lenguajes, y sin embargo se las arregla para ser incomprensible para cualquiera que conozca alguno de esos lenguajes. `(int)` parece C, pero `int` no existe. Los namespaces usan \. La sintaxis de arreglo `[key => value]`, es única entre los lenguajes que permiten mapas.
- El Tipado débil (conversión automática entre cadenas/números/otros) es tan complejo que cualquier esfuerzo menor que se le ahorre al programador realmente no vale la pena.
- Muy poca funcionalidad nueva es implementada mediante sintaxis nueva, la mayor parte se hace con funciones o cosas que parecen funciones. Excepto por el soporte de clases, que al parecer sí mereció un conjunto completamente nuevo de operadores y palabras reservadas.
- Algunos de los problemas listados tienen una solución profesional… si están dispuestos a pagarle a Zend para que arreglen su propio lenguaje de programación de código abierto.
- Hay mucha acción controlada a distancia. Consideremos este código, tomado de la documentación de PHP.

```
@fopen('http://example.com/not-existing-file', 'r');
```

¿Qué hace?

    - Si PHP fue compilado con `--disable-url-fopen-wrapper`, no va a funcionar. (La documentación no dice que significa "no va a funcionar": ¿Retorna `null`? ¿Lanza una excepción?) Noten que esta bandera fue eliminada en PHP 5.2.5.
    - Si `allow_url_fopen` está deshabilitado en `php.ini`, tampoco funciona, para evitar ataques de inclusión remota de archivos (RFI). Que como vulnerabilidad, debería estar deshabilitado por defecto.
    - Debido a la `@`, la advertencia sobre el fichero que no existe no será impresa.
    - Pero sí se imprime si `scream.enabled` está habilitado en php.ini.
    - O si `scream.enabled` se configura manualmente con `ini_set`.
    - Pero no si el nivel correcto de `error_reporting` no está configurado.
    - Si finalmente se logra imprimir, donde se imprime exactamente depende de `display_errors`, de nuevo en `php.ini`. O `ini_set`.

  No es posible decidir cómo esta función inofensiva se va a comportar sin consultar banderas de compilación, configuraciones a nivel del servidor, y configuraciones de la aplicación. Y todo esto es comportamiento por **defecto**.

- El lenguaje está lleno de estados globales e implícitos. `mbstring` usa un conjunto de caracteres global.
- `func_get_arg` y sus amigos parecen funciones regulares, pero operan en la función que se está ejecutando actualmente. El manejo de errores/excepciones utiliza valores globales por defecto. `register_tick_function` utiliza una función global para correr cada tick. ¿Por qué?
- No hay ningún tipo de soporte para programación multihilo. (Dado lo anterior, era de esperarse.) Combinado esto con la falta de una funcionalidad `fork` (mencionada más abajo), la programación paralela es muy difícil.
- PHP 4 y 5 solo pueden ser usados con el modelo `mpm_prefork` de Apache2 no con el `mpm_worker`. Esto significa que PHP limita las opciones de rendimiento con el servidor Apache. Al parecer sí es posible usarlo con `mpm_worker` si no se usa el módulo `mod_php` directamente, sino a través de FastCGI. Sin embargo, hasta el 2006 hubo al menos un error que impedía el uso confiable de FastCGI en este contexto. Y en general, podemos remitirnos a una presentación dada por Cal Henderson sobre el uso de PHP en Flickr, donde dice que PHP "pierde memoria como una criba", lo cual constituye un problema mayor para hacer que PHP funcione con FastCGI, SCGI o cualquier otro modelo de ejecución con hilos.
- Algunas partes de PHP están prácticamente diseñadas para producir código defectuoso.
    - `json_decode` retorna `null` para cualquier entrada inválida, a pesar de que `null` también es un objeto JSON perfectamente válido. Esta función no es confiable a no ser que se llame `json_last_error` cada vez que se use.
    - `array_search`, `strpos`, y otras funciones similares retornan 0 si se encuentra el puntero en la posición cero, pero `false` si no lo encuentran en absoluto.

Hablemos un poco más sobre este último tema.

En C, las funciones como `strpos` retornan `-1` si el elemento no se encuentra. Si no se chequea este caso, encuentran memoria basura y el programa explota...a veces. Recuerden que es C y nadie puede estar seguro de nada.

Por ejemplo, en Python los métodos `.index` equivalentes lanzan excepciones si el elemento no se encuentra. Si no chequean este caso, el programa explotará.

En PHP, estas funciones retornan false. Si se usa `FALSE` como índice, o se hace cualquier otra cosa que no sea compararlo usando `===`, PHP lo convierte silenciosamente en 0. El programa no explota, en vez de eso simplemente hará lo **incorrecto** sin **advertencia**, a no ser que recuerden poner el código boilerplate correcto en cada llamada a `strpos` y otras ciertas funciones.

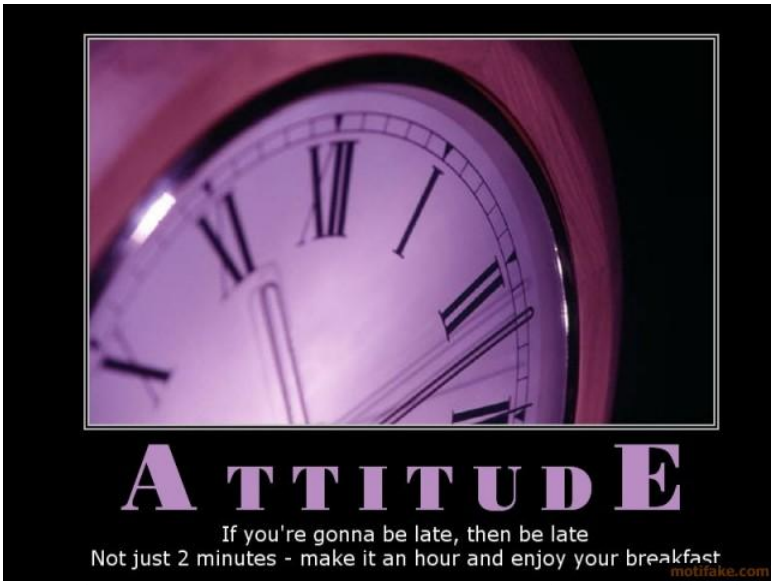
¡Esto es malo! Los lenguajes de programación son herramientas, se supone que trabajen **conmigo**. Pero aquí PHP ha creado una trampa sutil para que yo caiga, y tengo que vigilarla Aun haciendo operaciones tan simples como operaciones de cadenas y comparaciones de igualdad. PHP es un campo **minado**.

He oído muchas grandes historias sobre el intérprete de PHP y sus desarrolladores en muchos lugares. Historias de personas que han trabajado en el núcleo de PHP, lo han depurado y han interactuado con los desarrolladores principales. Ni una sola de estas historias es un elogio.

Versiones y desarrollo

- Desarrollo extremadamente lento. La existencia misma de un documento como este, donde fuentes del 2003 que documentan problemas de diseño fundamentales con el lenguaje todavía son válidas da que pensar. En las palabras de Jeff Atwood:
  "Recuerdo mi primera experiencia con PHP en 2001. A pesar de mi cuestionable pedigrí en ASP y Visual Basic, revisar una lista alfabética de funciones de PHP fue suficiente para alejarme durante años (…) no parece que las cosas hayan mejorado mucho desde entonces.

  (…) el diseño de lenguajes es muy duro. Hay una razón por la cual varios de los más famosos científicos de la computación también son diseñadores de lenguajes. Y es una lástima que ninguno de ellos haya tenido la oportunidad de trabajar en PHP. Por lo que se ve, PHP no es tanto un lenguaje como una colección aleatoria de cosas arbitrarias, una explosión virtual en la fábrica de funciones y palabras reservadas. Téngase en cuenta que esto lo dice un hombre criado en BASIC (…) así que no soy un extraño en este tema.



"Recuerdo mi primera experiencia con PHP en 2001. A pesar de mi cuestionable pedigrí en ASP y Visual Basic, revisar una lista alfabética de funciones de PHP fue suficiente para alejarme durante años (…) no parece que las cosas hayan mejorado mucho desde entonces."

Por supuesto, estos son viejas noticias. ¿Qué tan viejas? Antiguas. Antiguas como Internet Explorer 4. La internet está llena de artículos anti PHP – prácticamente se me acabaron las pestañas del navegador tratando de abrirlos todos."

Es como si los desarrolladores de PHP no pudieran (o peor, no quisieran) arreglar los problemas conocidos del lenguaje. Como si pensaran que no se puede hacer mejor que como está ahora, que no es necesario. Claro, que cuando se ponen de veras a hacer correcciones al lenguaje, entonces hacen más daño que bien, como vemos en el punto siguiente.

- Existen tres versiones incompatibles de PHP.
  Ha habido dos cambios incompatibles grandes en la base de código que han hecho que cierta cantidad del mismo sea imposible de portar hacia delante o hacia atrás.
  En detalle, los cambios han sido:

  - De PHP 4.3 a PHP 4.4 el comportamiento de las referencias ha sido cambiado para evitar pérdidas de memorias.

    Esto significa que este código, perfectamente razonable:

    ```
    function &myFactoryMethod() {
      return &new ProtectedClass();
    }
    ```

    deja de funcionar. En vez de ello, para hacer lo mismo, ahora tienes que escribir:

    ```
    function &myFactoryMethod() {
       return $var = &new ProtectedClass();
    }
    ```

    Aunque los scripts pequeños no suelen usar este patrón Factory (o ninguno) cualquier aplicación de tamaño mediano con buen diseño sí los aplica.

  - De PHP 4 a PHP 5 muchos conceptos clave del lenguaje cambiaron para soportar OOP.

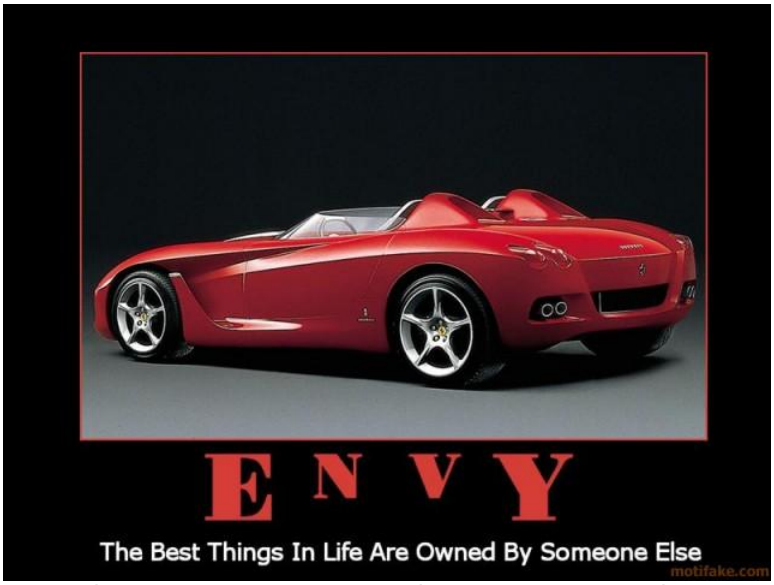
    No solo se introdujeron nuevas funcionalidades, sino que se cambió mucho comportamiento. Las llamadas por referencia ahora son el estándar, se introdujeron excepciones, se reservaron nuevas palabras, se adicionó visibilidad de atributos y métodos. Aunque Zend asegura que "la mayoría de los scripts funcionarán sin problemas con el nuevo modelo de objetos" lo cierto es que hasta el 2006 muy pocos sistemas habían sido portados de PHP 4 a 5, a pesar de que PHP 5 es una mejora inmensa para los programadores del lenguaje. Desafortunadamente muchos sistemas mantienen PHP 4 instalado, debido a que no funcionan con PHP 5, en contra de lo dicho por Zend.

Un problema más pequeño que no debe quedar sin mencionar es que la idea, generalmente buena, de hacer que todo sea sencillo proveyendo acceso directo a funciones como las mysql_* también fue en contra de la compatibilidad con MySQL 4.1+ cuando este introdujo nuevas funcionalidades binarias. Cualquier script que no use las funciones mysqli_* o una abstracción apropiada como ADODB o PEAR-DB es vulnerable y debe ser actualizada a la última versión de la base de datos.

- Licencias confusas

  Se podría pensar que PHP es libre y que los módulos de PHP mencionados en el manual también lo son. Y estaríamos equivocados. Por ejemplo, para generar un fichero PDF en PHP se pueden encontrar dos módulos en el manual: PDF o ClibPDF. Pero ambos tienen licencias comerciales. Así que para cada módulo, hay que estar seguro que se está de acuerdo con su licencia particular. O buscar una solución de terceros, que no se menciona en la documentación. (Como FPDF, por ejemplo).

- No existen funcionalidades originales: PHP, por sus mismas características de desarrollo lento, comunitario y no tener tanto potencial de desarrollo como las demás plataformas empresariales a lo único que atina es a reproducir las prácticas probadas de las demás tecnologías. El mayor ejemplo es, por supuesto, la OOP… implementada literalmente décadas después de que ya existiera y estuviera en uso por los lenguajes dominantes. Igualmente con las funcionalidades de reflection, abstracción de acceso a datos, namespaces, referencias, iteradores…
  Y sin hablar de los frameworks, que siempre están a la zaga de los lenguajes que los utilizan intensamente al punto de incorporar sus buenas prácticas dentro de sus IDEs e incluso del lenguaje mismo, como los EJB de Java o el Entity Framework de C#.



PHP a lo único que atina es a reproducir las prácticas probadas de las demás tecnologías.

Operadores

- == es inútil.
  - No es transitivo. `"foo" == TRUE`, y `"foo" == 0`… pero por supuesto `TRUE!= 0`.

    `(string)"false" == (int)0` es *true*

    Nótense los casteos explícitos. Incluso en un lenguaje con tipos dinámicos esto no debería ocurrir, porque se han determinado los tipos “estáticamente” mediante casting.

Lo opuesto de esta sentencia también evalúa a true: `(boolean)false == (string)"0"`. Esto es un problema de la conversión de tipos de PHP, porque cualquier cadena que no esté vacía se interpreta como true.

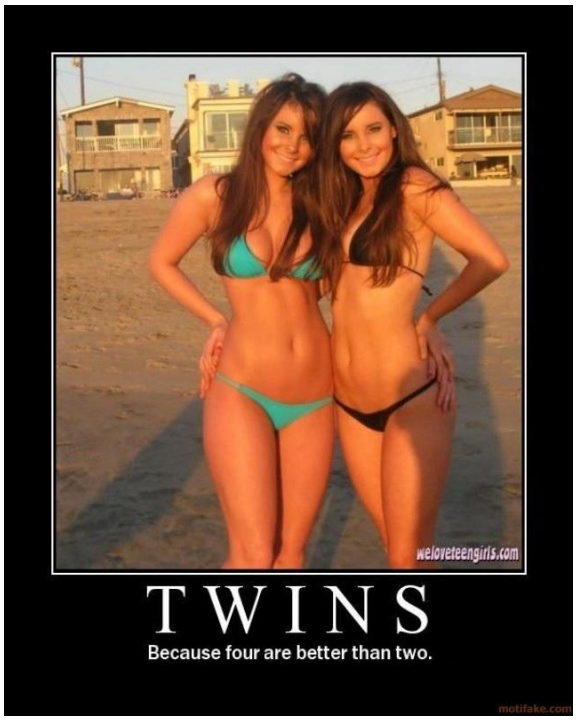
Por supuesto este problema tiene una solución, mediante el concepto (dañado) de identidad de PHP. Miremos el código siguiente:

```
if ("false" == true) echo "true\n";
// => true

if ("false" == false) echo "true\n";
// => false

if ("false" == 0) echo "true\n";
// => true, wtf

if (false == 0) echo "true\n";
// => true, como se espera

// así que, "false" = true && "false" = 0.Entonces "false" es
true y
// false es false y no hemos discutido aún la identidad.

if ((string)"false" === (int)0) echo "true\n";
// => false, ...ok...

if ("0" === 0) echo "true\n";
// => false

if ("false" === false) echo "true\n";
// => false

if ((int)"0" === 0) echo "true\n";
// => true, con coerción de tipos
```

Esto significa que si se quieren comparar predeciblemente dos variables y no se tiene control sobre su entrada (por ejemplo, si una es una variable de usuario) tienes que compararlos por identidad, y luego convertirlos explícitamente, conociendo sus tipos, para obtener un resultado predecible. Lo cual significa que "false" es `0` o `false`, pero también que "false" es `true`… a veces.

- `==` convierte a números cuando es posible (`123 == "123foo"`… aunque `"123"!= "123foo"`), lo cual significa que convierte a float cuando es posible. Así que es posible que cadenas hexadecimales, como por ejemplo, contraseñas, pueden ocasionalmente comparar como `TRUE` aunque no lo sean. Ni siquiera JavaScript hace esto.
- Por la misma razón, `"6" == " 6"`, `"4.2" == "4.20"`, y `"133" == "0133"`. Pero nótese que `133!= 0133`, porque `0133` es octal. ¡*Pero* `"0x10" == "16"` y `"1e3" == "1000"`!
- `===` compara valores y tipos… excepto con objetos donde `===` solo es `TRUE` si ambos operadores son realmente el mismo objeto. Para los objetos, `==` compara tanto valor como tipo… que es lo que hace `===` para cualquier otro tipo de datos.

- Las comparaciones no son mucho mejores.
    - Ni siquiera son consistentes: `NULL < -1`, *y* `NULL == 0`. EL ordenamiento es por tanto no determinista, depende del orden en el que el algoritmo de ordenamiento compare los elementos.
    - Los operadores de comparación tratan de ordenar arreglos de dos maneras: primero por longitud, después por elementos. Si tienen el mismo número de *elementos*, pero distintos *conjuntos de llaves*, son incomparables.
    - Los objetos siempre se comparan como más grandes que cualquier otra cosa… excepto otros objetos, con los cuales no son ni mayores ni menores.
    - Para un operador `==` con más seguridad de tipo, existe `===`. Para un operador `<` con más seguridad de tipo, tenemos… nada. `"123" < "0124"`, siempre, sin importar lo que hagan. Castear no ayuda tampoco.



Los objetos siempre se comparan como más grandes que cualquier otra cosa… excepto otros objetos, con los cuales no son ni mayores ni menores.

- A pesar de la locura anterior, y el rechazo explícito a los operadores de cadena y numéricos de Perl, PHP no sobrecarga `+`. `+` siempre es adición, y . siempre es concatenación.
- El operador índice `[]` también puede representarse como `{}`.
    - `[]` se puede usar en cualquier variable, no solo cadenas o arreglos. Simplemente regresa `null` y no da ni advertencias.
- `[]` no sirve para elegir conjuntos, solo se puede usar para sacar elementos individuales. `foo()[0]` es un error de sintaxis. (Arreglado en PHP 5.4.)
- Los operadores || y && en lenguajes sin retraso mental regresan el resultado del operador izquierdo o el resultado del operador derecho.

  En PHP, 5 || 4 retorna 1 y 0 || 6 también retorna 1.

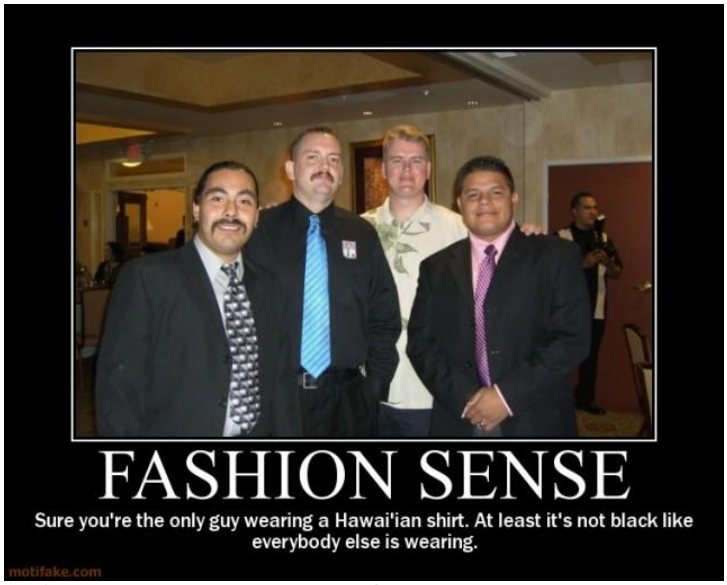
  O sea, que el uso que algunas personas dan a || como un operador para valores por defecto:

  ```
  $foo = $bar || "default value";
  ```

  no puede hacerse en PHP.

- A diferencia de **literalmente** todos los demás lenguajes con un operador similar `?:` es asociativo **a la izquierda**. De manera que:

```
  $arg = 'T';
  $vehicle = ( ( $arg ==
'B' ) ? 'bus' :
                ( $arg ==
'A' ) ? 'airplane' :
                ( $arg ==
'T' ) ? 'train' :
                ( $arg ==
'C' ) ? 'car' :
                ( $arg ==
'H' ) ? 'horse' :
                'feet' );
  echo $vehicle;
```



*A diferencia de **literalmente todos** los demás lenguajes con un operador similar, ?: es asociativo **a la izquierda**.*

imprime `horse`. Esto fue seguramente un accidente. El comportamiento correcto se podía restaurar con un solo cambio sencillo en el parser, pero ahora ya es demasiado tarde (rompería la compatibilidad hacia atrás).

Variables

- No hay manera de declarar una variable. Las variables que no existen se crean con valor nulo en su primer uso, provocando errores interesantes si no se tiene cuidado con su ámbito.
- Las variables globales necesitan una declaración `global` antes de poder ser usadas. Esta es una consecuencia natural de lo anterior, así que sería perfectamente razonable… excepto que las globales no pueden ser siquiera leídas sin una declaración específica. PHP silenciosamente crea una variable local con el mismo nombre. No conozco ningún otro lenguaje con problemas de ámbito semejantes.
- No existen referencias. Lo que PHP llama referencias son en realidad alias, no hay referencias al estilo Perl ni pass-by-object como en Python. Las referencias son sintácticamente complejas y están implementadas "malamente", para ser corteses. PHP 5 resuelve algo de esta mala implementación con un cambio incompatible.

  Esto significa que cualquier manejo de objetos, arreglos o mapas donde se necesite alto rendimiento requiere grandes cantidad de azúcar sintáctica para evitar el manejo de copia por defecto y paso por valor de PHP 4. En PHP 5, los objetos se pasan por referencia, pero estas referencias son en realidad alias, lo cual significa que aunque se cree una instancia anónima, si esta no se asigna a una variable el recolector de basura no la encuentra.

```
function &myFactoryMethod() {
  return $var = &new ProtectedClass();
}
$xyz =& myFactoryMethod();
```

```
// aqui $xyz no es una referencia al objeto retornado, sino un ALIAS
para $var
```

¿Por qué no crearon nombres automáticamente para las variables anónimas? No tengo idea. <sarcasm> Quizás pensaron que sería mal diseño</sarcasm>.

- Las referencias infectan una variable como ninguna otra cosa en el lenguaje. PHP es dinámicamente tipado, así que las variables generalmente no tienen tipo… excepto las referencias, que adornan las definiciones de función, sintaxis de variables y asignaciones. Una vez que una variable se convierte en referencia (lo que puede pasar en cualquier lugar) está atascada como una referencia. No existe una manera obvia para detectar esta situación, y eliminar la referencia requiere eliminar la variable por completo.
- Bueno, no en realidad. Están los tipos SPL, que también infectan variables: `$x = new SplBool(true); $x = "foo";` falla. Es justo como el tipado estático, como pueden ver.
- Se puede hacer una referencia a una llave que no existe dentro de una variable indefinida (que se convierte en un arreglo). Usar un arreglo que no existe normalmente da una advertencia, pero esto no lo hace.
- Las constantes se definen mediante una llamada a función que toma una cadena como argumento, antes de esta llamada, no existen. (Esto puede ser una copia del comportamiento del `use constant` de Perl.)
- Los nombres de variable son sensibles a mayúsculas. Las funciones y los nombres de clase no.

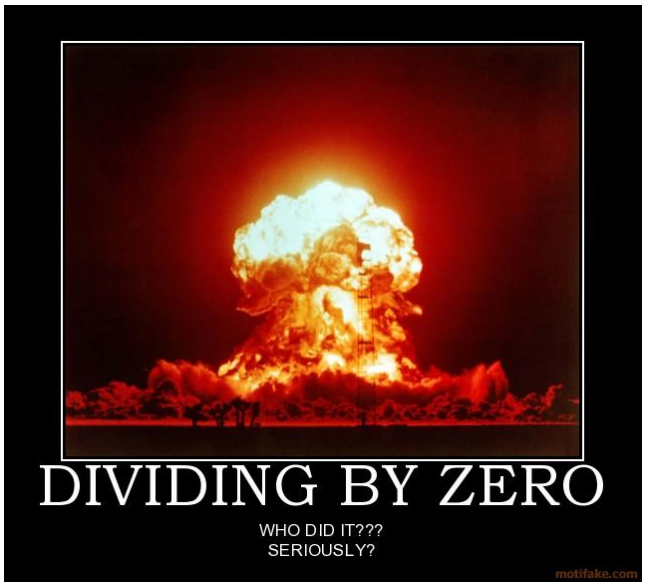
Construcciones del lenguaje

- `array` y algunas docenas de construcciones similares no son funciones. `array` por sí mismo no significa nada, `$func = "array"; $func();` no funciona.
- Desempacar un arreglo puede hacerse con `list($a, $b) = ...`, donde `list()` es una sintaxis parecida a función, como `array`. Por qué esto no tiene una sintaxis real, o por qué el nombre es tan confuso, no lo sé.
- `(int)` está obviamente diseñado para parecer C, pero es un token único, no existe nada llamado `int` en el lenguaje. Pueden probarlo, no solo `var_dump(int)` no funciona, lanza un error de parseo debido a que el argumento se parece al operador de casteo.
- `(integer)` es un sinónimo de `(int)`. También existen `(bool)`/`(boolean)` y `(float)`/`(double)`/`(real)`.
- Existe un operador `(array)` para castear a un arreglo, y `(object)` para castear a objeto. Suena loco, pero es casi útil: Se puede usar `(array)` como un argumento de función que es o bien un solo elemento o una lista, y los trata igual. Pero esto no se puede hacer con confianza, porque si se pasa un objeto, castearlo a arreglo lo convierte en un arreglo que contiene todos los atributos del objeto. Castear un arreglo a objeto produce la operación inversa.
- `include()` y sus amigos son básicamente como el `#include de C`: descargan otro archivo de contenido dentro del archivo llamador. No existe un sistema de módulos.
- No hay funciones ni clases locales ni anidadas. Solo hay globales. Incluir un archivo adiciona sus variables al ámbito de la función que lo llama (y de paso le da acceso en ese fichero a tus variables), pero las clases y las funciones siempre se adicionan al ámbito global.
- Adicionar al final de un arreglo se hace con `$foo[] = $bar`.
- `echo` no es una función, sino una construcción especial del lenguaje.

- `empty($var)` no es una función tan extremadamente que cualquier cosa que no sea una variable. Por ejemplo, `empty($var || $var2)`, es un error de parseo. ¿Por qué tiene que saber el parser acerca de `empty`?
- Existe sintaxis redundante para definir bloques: `if (...): ... endif;`, etc.

Manejo de errores

- El único operador especial de PHP es `@` (de hecho copiado de DOS), que **acalla** los errores.
- Los errores de PHP no dan stack traces. Tiene que instalarse un manejador para generarlas (Excepto para errores fatales, lean más adelante.) o utilizar código boilerplate para imprimirlas. Con XDebug se corrige esto en las versiones más modernas de 5.x, pero cuando los errores son compuestos o numerosos se muestran de manera confusa directo a la página.
- Los errores de parseo generalmente solo lanzan el estado del parseo y nada más, haciendo que una comilla sea muy difícil de depurar.
- El parser de PHP se refiere a, por ejemplo a `::` internamente como `T_PAAMAYIM_NEKUDOTAYIM`, y al operador `<<` como `T_SL`. Digo internamente, pero como arriba, esto es lo que se muestra al programador si `::` o `<<` aparecen en el lugar incorrecto.



Los errores de PHP no dan stack traces. Tiene que instalarse un manejador para generarlas

- La mayor parte del manejo de errores es escribiendo una línea a un archivo de log que nadie lee y siguiendo con el programa.
- `E_STRICT` existe, pero no parece realmente prevenir mucho, y no hay documentación sobre que hace realmente.
- `E_ALL` incluye todas las categorías de error —excepto `E_STRICT`. Es mejor usar `error_reporting(-1)` en vez de `error_reporting(E_ALL)`. Esto sí habilita `E_STRICT`, `E_NOTICE`, etc. Me encanta como PHP usa su propia definición de "todo". (Arreglado en 5.4.)
- Salvajemente inconsistente con lo que se permite y lo que no. No se cómo se aplica `E_STRICT` aquí, pero estas cosas se puede hacer:
    - Acceder a una propiedad que no existe de un objeto, ej. `$foo->x`. (advertencia)
    - Usar una variable como un nombre de función o de variable, o de clase. (silencioso)
    - Tratar de usar una constante indefinida. (información)
    - Tratar de acceder a una propiedad de algo que no es un objeto. (información)
    - Tratar de usar un nombre de variable que no existe. (información)
    - `2 < "foo"` (silencioso)
    - `foreach (2 as $foo);` (advertencia)

    Y estas no se pueden hacer:

    - Tratar de usar una constante de clase no existente. Ej. `$foo::x`. (error fatal)

  - Usar una constante de cadena como un nombre de función o nombre de variable, o nombre de clase. (error de parseo)
  - Tratar de llamar una funcion indefinida. (error fatal)
  - No poner un punto y coma en la última línea de un bloque o archivo. (error de parseo)
  - Usar `list` y otras entidades del lenguaje como nombres de método. (error de parseo)
  - Utilizar directamente el valor de retorno de una función, ej., `foo()[0]`. (error de parseo, arreglado en 5.4)

  Hay algunos otros buenos ejemplos de errores raros de parseo en otros lugares de esta misma lista.

- El método `__toString` no puede lanzar excepciones. Si lo intentan, PHP… lanza una excepción (en realidad un error fatal, lo cual sería pasable, excepto que…)
- Los errores y las excepciones en PHP son bestias totalmente diferentes. No interactúan **en absoluto**.
  - Los errores de PHP (los internos y las llamadas a `trigger_error`) no pueden ser capturados con `try`/`catch`.
  - Igualmente, las excepciones no ejecutan los manejadores de error instalados por `set_error_handler`.
  - En vez de eso, existe un `set_exception_handler` que maneja las excepciones no capturadas, porque envolver el punto de entrada del programa en un bloque `try/catch` es imposible en el modelo `mod_php`.
  - Los errores fatales (ej. `new ClassDoesntExist()`) no pueden ser capturados. Muchas cosas más o menos inocuas lanzan errores fatales, terminando forzosamente el programa por razones cuestionables. Las funciones de apagado todavía se ejecutan, pero no tienen `stack trace` y no pueden saber fácilmente si el programa termina normalmente o debido a un error.
- No existe un `finally`. A pesar de que la OOP y las excepciones fueron mayormente copiadas de Java, esta falta es deliberada porque `finally` "no tiene mucho sentido en el contexto de PHP". ¿Qué?

Anotaciones

- Las anotaciones en PHP no existen. Sin embargo muchos frameworks y desarrolladores están tan necesitados de esta configuración por metadatos que se las han inventado, escribiendo pseudo-anotaciones en los comentarios especiales para documentación. Esta simple decisión reduce el rendimiento, la flexibilidad y la portabilidad de todos los sistemas que la usan, además de crear otra serie de interesantísimos e impredecibles problemas de depuración y aprendizaje. También, son primitivas (el autowiring por tipo y en colecciones es prácticamente imposible, en contraste con Spring en Java) y poco elegantes.

Funciones

- Las llamadas a función son, aparentemente, costosas.
- Este código funciona: `$x = &collector();`

Este código produce un error fatal: `array_push(&collector(), 'foo');`
Este código funciona: `array_push(collector(), 'foo');`

Este comportamiento no puede ser deducido de ningún conocimiento previo sobre las referencias en PHP ni otro lenguaje, debido a que se produce por un caso especial manejado por el intérprete de PHP, específicamente para los casos en que las funciones del sistema interactúan con funciones que devuelven referencias.

- Como se ha mencionado antes, muchas cosas que parecen funciones o que parece que deberían comportarse como funciones son en realidad construcciones particulares del lenguaje, así que nada que trabaje con funciones trabajará con ellas.
- Los argumentos de funciones pueden tener "sugerencias de tipo", que básicamente es lo mismo que tipado estático. Pero no se puede requerir que un argumento sea un `int` o `string` u `object` o cualquier otro tipo principal, aunque todas las funciones del sistema usen este tipado, probablemente porque int no existe en PHP. Tampoco se pueden usar los falsos tipos que usan mucho las funciones del sistema: `mixed`, `number`, o `callback`. (`callable` se permite en PHP 5.4.)

  Como resultado, esto:

  ```
  function foo(string $s) {}

  foo("hello world");
  ```

  produce el error:

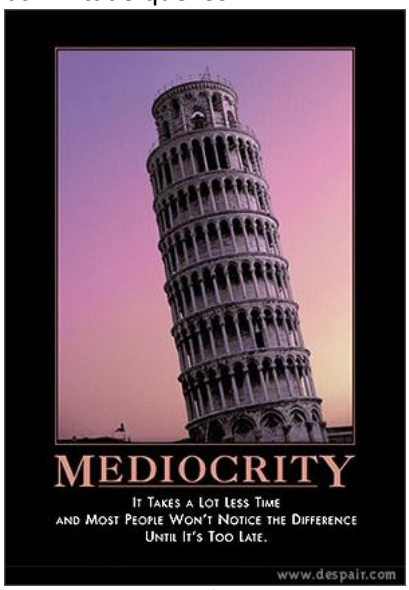
  ```
  PHP Catchable fatal error:  Argument 1 passed to foo() must be
an instance of string, string given, called in...
  ```

  - Podrán notar que la "sugerencia de tipo" dada no tiene que realmente existir, no hay una clase `string` en este programa. Sí tratan de usar `ReflectionParameter::getClass()` para examinar la sugerencia de tipo dinámicamente, **entonces** se queja de que la clase no existe, haciendo que sea imposible devolver realmente el nombre de la clase.
  - No se pueden usar estas sugerencias en los valores de retorno de función.
- Pasar los argumentos de la función actual a otra función (una tarea bastante común) se hace mediante `call_user_func_array('other_function', func_get_args())`. Pero `func_get_args` lanza un error fatal en tiempo de ejecución, quejándose de que no puede usarse como un parámetro de función. ¿Cómo y por qué razón es esto un error siquiera? (Arreglado en PHP 5.3.)
- Las closures requieren que se nombre explícitamente todas las variables que van a utilizar. ¿Por qué el intérprete no puede encontrar estos valores por sí mismo? Entorpece el uso de la funcionalidad en sí misma. Pero no puede ser de otra forma, porque usar una variable de cualquier manera la crea, a no ser que específicamente se diga lo contrario.
- Las variables cerradas (trabajadas con closures) se pasan con la misma semántica que los argumentos de función. O sea, los arreglos y las cadenas se pasan por valor. A no ser que usen `&`.

- Dado que las variables cerradas son argumentos pasados automáticamente y no existen ámbitos anidados, una closure no puede referirse a métodos privados, incluso si se define dentro de una clase. (Posiblemente arreglado en 5.4, no queda claro.)
- No existen argumentos nombrados para las funciones. Rechazados específicamente por los desarrolladores debido a que "permiten código más desordenado".
- Los argumentos de función con valores por defecto pueden aparecer antes que aquellos que no tienen, incluso a pesar de que la documentación especifica que esto es raro e inútil. ¿Entonces por qué lo permiten?
- Los argumentos extra de una función se ignoran (excepto en las funciones del sistema, que lanzan un error). Los argumentos faltantes se asumen nulos.
- Las funciones con argumentos variables requieren el uso de funciones como `func_num_args`, `func_get_arg`, y `func_get_args`. No existe sintaxis dedicada para estas cosas.

OOP

- Las partes procedurales de PHP están diseñadas como C, pero la parte objetual está diseñada como Java. No puedo sobreestimar lo confuso que es esto. El sistema de clases está diseñado alrededor del nivel *más bajo* de lenguaje Java, que es mucho más limitado que los contemporáneos de PHP, y esto me sorprende.
    - Todavía no he encontrado una función global con una letra mayúscula en su nombre, sin embargo funciones importantes del sistema utilizan notación camellada en sus nombres de método y tienen accesores con estilo Java como `getFoo`.
    - Perl, Python, y Ruby todos tienen algún concepto de acceso al código mediante "propiedades". PHP solo tiene el molesto `__get` y sus amigos. (La documentación inexplicablemente se refiere a tales métodos especiales como "sobrecarga".)
    - Las clases tienen algo semejante a la declaración de variables (`var` y `const`) para los atributos de clase, mientras que la parte procedural del lenguaje no lo tiene.
    - A pesar de la gran influencia de C++/Java, donde los objetos son bastante opacos, PHP normalmente trata a los objetos como hashes sofisticados. Por ejemplo, el comportamiento por defecto de `foreach ($obj as $key => $value)` es iterar sobre todos los atributos accesibles del objeto.



El sistema de clases está diseñado alrededor del nivel más bajo de lenguaje Java, que es mucho más limitado que los contemporáneos de PHP, y esto me sorprende.

- Las clases no son objetos. Cualquier tipo de meta programación tiene que referirse a ellas utilizando una cadena, como las funciones.
- Los tipos por defecto no son objetos, y no se puede, a diferencia de Java, C# o Perl, hacer que se puedan manejar como objetos de ninguna manera.
- `instanceof` es un operador, a pesar de que las clases son una adición posterior y la mayor parte del lenguaje está construido sobre funciones y cosas que parecen funciones. ¿Influencia de Java? ¿Las clases no son entidades de primer nivel? No tengo idea.
    - Pero *existe* una función `is_a`. Con un argumento opcional para especificar si queremos permitir que el objeto sea una cadena que contenga el nombre de una clase.

    - `get_class` es una función; no hay un operador `typeof`. Lo mismo para `is_subclass_of`.
    - Esto no funciona con los tipos por defecto (de nuevo, `int` no existe). Para eso necesitan usar `is_int`, etc.
    - También, el lado derecho tiene que ser una variable o cadena literal, no puede ser una expresión. Esto causa… un error de parseo.
- ¡¿`clone` es un operador?!
- Los atributos de objetos son `$obj->foo`, pero los atributos de clase son `Class::$foo`. (`$obj::$foo` tratará de convertir en cadena a `$obj` y usarlo como un nombre de clase.) Los atributos de clase no pueden accederse desde los objetos, los namespaces están completamente separados, haciendo que los atributos de clases sean inútiles para el polimorfismo. Los métodos de clase, por supuesto, están exentos de esta regla y pueden llamarse como cualquier otro método. (Me dicen que C++ también lo hace así. C++ *no es* un buen ejemplo de OOP.)
- También, un método de una instancia se puede llamar estáticamente desde la clase, pero si se hace desde dentro de otro método, la llamada se trata como si fuera sobre el `$this` actual. Tomemos este ejemplo sacado de la documentación de PHP 5.3.x:

```
<?php
class A
{
 function foo()
 {
  if (isset($this)) {
   echo '$this está definida (';
   echo get_class($this);
   echo ")\n";
  } else {
   echo "\$this no está definida.\n";
  }
 }
}

class B
{
 function bar()
 {
  // Nota: la siguiente línea arrojará un Warning si E_STRICT está habilitada.
  A::foo();
 }
}

$a = new A();
$a->foo();

// Nota: la siguiente línea arrojará un Warning si E_STRICT está habilitada.
A::foo();
$b = new B();
```

```
$b->bar();

// Nota: la siguiente línea arrojará un Warning si E_STRICT está habilitada.
B::bar();
?>
```

El resultado del ejemplo sería:

```
$this está definida (A)
$this no está definida.
$this está definida (B)
$this no está definida.
```

También, cualquier modificación que se realice sobre los atributos de la clase A mientras está trabajándose con el `$this` de la clase B provoca que la instancia de B que se está usando gane los atributos de A, y los mantenga como propios hasta que sea destruida. Si les parece complejo no los culpo, lo es.
Un efecto secundario de esta locura es que si usan `$this` en los métodos de una clase están produciendo un código que funciona si es invocado desde objetos de la clase, y no funciona si es invocado estáticamente desde objetos de otra clase. Y no existe manera de saber desde donde se está invocando si `display_errors` está desactivado. O si no se especifica `E_ALL o E_STRICT` como niveles de error, los cuales están recomendados para servidores de desarrollo pero no de producción (buena suerte actualizando una aplicación en un servidor de Hosting remoto donde no hay control del `php.ini`). Básicamente, este método funciona a veces y a veces no. Que viva la extensibilidad...

- `new`, `private`, `public`, `protected`, `static`, etc. ¿Tratando de ganarse los desarrolladores de Java? Está claro que esto es más bien gusto personal, pero no sé para qué un lenguaje de tipado dinámico necesita estas cosas. En C++ la mayor parte de ello es necesaria solo para resolución de nombres en tiempo de compilación. También debido al punto anterior `static` es prácticamente inútil y solo sirve para determinar buenas prácticas.
- `self` es una palabra reservada que sustituye al nombre de clase en las llamadas estáticas dentro de la clase. Además de que estas llamadas estáticas son peligrosas porque pueden usar un `$this` de objetos que no son instancias de la clase actual, esta palabra es totalmente inútil. Nótese que también se puede usar el nombre de la clase para las llamadas (`self::foo()` es lo mismo que `Bar::foo()` donde `Bar` es el nombre de la clase en ambos casos). Entonces, ¿Por qué no utilizar siempre el nombre de la clase y evitar el uso de otra palabra reservada más?
- PHP tiene soporte de primer nivel para clases abstractas, aquellas clases que no pueden ser instanciadas. El código de lenguajes similares logra este mismo efecto lanzando excepciones en el constructor.
- Las subclases no pueden sobrescribir métodos privados. Los métodos públicos sobrescritos no pueden ver, ni hablemos de llamar, a los métodos de la superclase. Problemático para mocks de prueba y otras situaciones.
- Los métodos no pueden ser llamados, por ejemplo, “list”, porque `list()` es una sintaxis especial (no es una función) y el parser se confunde. No hay ninguna razón por la cual esto debería ser ambiguo, y parchear la clase funciona perfectamente. (`$foo->list()` no es un error de sintaxis.)
- Si una excepción es lanzada al evaluar los argumentos de un constructor (ej., `new Foo(bar())` y `bar()` provoca el error), el constructor no se invoca, pero el *destructor* sí. (Arreglado en PHP 5.3.)

- Las excepciones en `__autoload` y los destructores causan errores fatales. (Arreglado en PHP 5.3.6. Ahora un destructor puede lanzar una excepción literalmente en cualquier lugar, dado que es llamado en el momento en que el conteo de referencias llega a 0. Hum.)
- No hay constructores ni destructores. `__construct` es en realidad un inicializador, como el `__init__` de Python. No hay método que se pueda llamar en una clase para reservar memoria y crear un objeto.
- No hay inicializador por defecto. Llamar `parent::__construct()` si la clase padre no define su propio `__construct` es un error fatal.
- OOP trae una interface para iteradores que algunas partes del lenguaje respetan (ej., `for...as`) pero que nada que venga por defecto (como los arreglos) la implementa en realidad. Si quieren iterar por un arreglo, tienen que envolverlo en un `ArrayIterator`. No hay formas por defecto de encadenar o dividir o de cualquier otra manera trabajar con iteradores como objetos de primer nivel.
- Las interfaces como `Iterator` reservan una cierta cantidad de buenos métodos sin prefijo. Si quieren una clase iterable (que no sea el comportamiento por defecto de iterar por todos los atributos) y quieren usar un nombre común de método, como `key` o `next` o `current`, mala suerte.
- Las clases pueden sobrecargar la conversión a cadenas, pero no la conversión a números ni a ninguno de los otros tipos por defecto del lenguaje.
- Las cadenas, números y arreglos todos tienen conversión de cadena; de hecho el lenguaje depende fuertemente de esta propiedad. Las funciones y las clases *son* cadenas. Sin embargo, tratar de convertir un objeto del sistema o de usuario a una cadena causa un error si no implementa `__toString`. Incluso `echo` se convierte en una fuente potencial de errores.
- No hay sobrecargas para comparar igualdad o para ordenar.
- Las variables estáticas dentro de métodos de instancia son globales, comparten el mismo valor para todas las instancias de esa clase.

Librería estándar

- No existe un sistema de módulos. Se pueden compilar extensiones de PHP, pero cuales están cargadas se especifica en el php.ini, y las opciones son que una extensión existe (y se inyecta globalmente) o no existe en absoluto.
- Como los namespaces son una funcionalidad relativamente reciente, la librería estándar no está separada en absoluto. Hay miles de funciones en el namespace global.
- Partes de la librería son salvajemente inconsistentes entre sí:
    - Guión o no guión: `strpos`/`str_rot13`, `php_uname`/`phpversion`, `base64_encode`/`urlencode`, `gettype`/`get_class`
    - "to" versus 2: `ascii2ebcdic`, `bin2hex`, `deg2rad`, `strtolower`, `strtotime`
    - Objeto + verbo versus verbo + objeto: `base64_decode`, `str_shuffle`, `var_dump` versus `create_function`, `recode_string`

la librería estándar no está separada en absoluto. Hay miles de funciones en el namespace global.

    - Orden de los parámetros: `array_filter($input, $callback)` versus `array_map($callback, $input)`, `strpos($haystack, $needle)` versus `array_search($needle, $haystack)`
    - Confusión con los prefijos: `usleep` versus `microtime`
    - Alrededor de la mitad de las funciones de arreglo comienzan con `array_`. Las otras, no.
    - `htmlentities` y `html_entity_decode` son inversas una de otra, pero tienen convenciones de nombres completamente distintas.
- Tiene de todo. La librería incluye:
    - Uso de ImageMagick, uso de GraphicsMagick (que es un fork de ImageMagick), y un puñado de funciones para inspeccionar EXIF (lo cual ya hace ImageMagick).
    - Funciones para parsear bbcode, un tipo de marcado muy específico utilizado solo por un puñado de paquetes de foro particulares.
    - Demasiados paquetes de XML. `DOM` (OO), `DOM XML`, `libxml`, `SimpleXML`, "XML Parser", `XMLReader`/`XMLWriter`, y media docena más de siglas que no conozco. Seguramente hay alguna diferencia entre estas cosas. Si quieren pueden intentar averiguarlas.
    - Uso para solo dos paquetes particulares de tarjetas de crédito, SPPLUS and MCVE.
    - Tres maneras de acceder a MySQL: `mysql`, `mysqli`, y la abstracción `PDO`.

Influencia de C

Esto merece su propio apartado, porque es tan absurdo y sin embargo permea todo el lenguaje. PHP es un lenguaje de alto nivel y tipos dinámicos. Sin embargo una parte masiva de la librería estándar está conformada por wrappers finos sobre APIs de C, con los resultados siguientes:

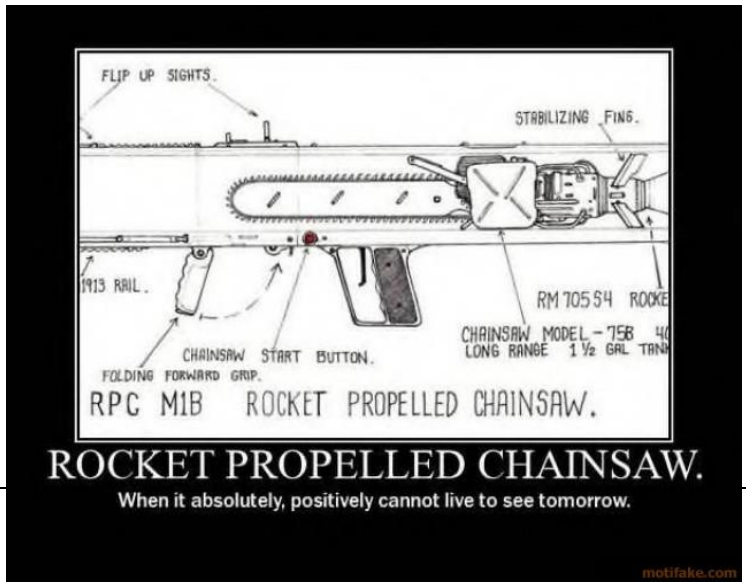
- Existen parámetros de salida, aunque PHP puede retornar hashes o argumentos múltiples sin esfuerzo.
- Al menos una docena de funciones para obtener el último error de un subsistema determinado, aunque PHP ha tenido excepciones durante los últimos ocho años.
- Abominaciones como `mysql_real_escape_string`, aunque tiene los mismos argumentos que la inútil `mysql_escape_string`, que solo existen porque son parte del API de MySQL C.
- Comportamiento global para funcionalidades no globales (como MySQL). Usar varias conexiones de MySQL aparentemente requiere pasar un manejador de conexión en cada llamada a función del API.
- Los wrappers son muy, pero muy finos. Por ejemplo, llamar `dba_nextkey` sin llamar `dba_firstkey` provoca un error de memoria.
- Existe un conjunto de funciones `ctype_*` (ej. `ctype_alnum`) que mapean las funciones para manejo de caracteres de C con nombres similares, en vez de, por ejemplo, usar `isupper`.

Genericidad

No existe. Si una función necesita hacer dos cosas ligeramente diferentes, PHP provee dos funciones. ¿Cómo se ordena al revés? En Perl, se puede hacer `sort { $b <=> $a }`. En Python, se puede hacer `.sort(reverse=True)`. En PHP, hay una función separada, `rsort()`.

- Funciones que buscan un error en C: `curl_error`, `json_last_error`, `openssl_error_string`, `imap_errors`, `mysql_error`, `xml_get_error_code`, `bzerror`, `date_get_last_errors`, etc.
- Funciones que ordenan: `array_multisort`, `arsort`, `asort`, `ksort`, `krsort`, `natsort`, `natcasesort`, `sort`, `rsort`, `uasort`, `uksort`, `usort`
- Funciones que encuentran texto: `ereg`, `eregi`, `mb_ereg`, `mb_eregi`, `preg_match`, `strstr`, `strchr`, `stristr`, `strrchr`, `strpos`, `stripos`, `strrpos`, `strripos`, `mb_strpos`, `mb_strrpos`, y sus variaciones que hacen reemplazos.
- Hay muchos alias, lo cual ciertamente no ayuda: `strstr`/`strchr`, `is_int`/`is_integer`/`is_long`, `is_float`/`is_double`, `pos`/`current`, `sizeof`/`count`, `chop`/`rtrim`, `implode`/`join`, `die`/`exit`, `trigger_error`/`user_error`, `diskfreespace`/`disk_free_space`…
- `scandir` retorna una lista de archivos dentro de un directorio dado. En vez de retornarlos en el orden en que existen en el directorio, la función los regresa ya ordenados. Y hay un argumento opcional para devolverlos en orden alfabético *inverso*. No había, aparentemente, suficientes funciones de ordenamiento. (PHP 5.4 adiciona un tercer valor para la dirección de ordenamiento, que deshabilitada el ordenamiento.)
- `str_split` rompe una cadena en pedazos del mismo tamaño, luego los une separados por un delimitador.
- Leer archivos requiere un grupo separado de funciones dependiendo del formato. Hay seis grupos separados de estas funciones, todas con diferentes APIs, para bzip2, LZF, phar, rar, zip, y gzip/zlib.
- Dado que llamar una función con un arreglo como parámetro es tan molesto (`call_user_func_array`), existen algunas parejas como `printf`/`vprintf` y `sprintf`/`vsprintf`. Estos hacen lo mismo, pero una función toma parámetros y la otra toma un arreglo de parámetros.

Texto



"gzgetss — Toma una línea de un archivo gz y elimina las etiquetas HTML." Me muero por conocer el conjunto de circunstancias que llevaron a concebir esta función.

- `preg_replace` con la bandera `/e` (eval) hace un reemplazo de las coincidencias en la cadena que se va a reemplazar, y luego la *evalúa*.
- `strtok` está aparentemente diseñado siguiendo su equivalente de C, lo cual es de por sí una mala idea por varias razones. No importa que PHP pueda fácilmente retornar un arreglo (mientras que esto no es cómodo de hacer en C).
- `parse_str` parsea cadenas de *consulta* URL, aunque el nombre no lo indica. Además actúa exactamente igual que `register_globals` al volcar la cadena en el ámbito local en forma de variables, a no ser que se le pase un arreglo para que lo llene. Por supuesto no tiene retorno.
- `explode` no divide con un delimitador vacío o faltante. Cualquier otra implementación en cualquier otro lenguaje hace algo útil por defecto en este caso, pero PHP tiene otra función separada, confusamente llamada `str_split` y que se describe como "convertir una cadena en un arreglo".
- Para formatear fechas existe `strftime`, que actúa como el API de C en lo que respecta al locale. También existe `date`, que tiene una sintaxis totalmente diferente y solo funciona en inglés.
- "`gzgetss` — Toma una línea de un archivo gz y elimina las etiquetas HTML." Me muero por conocer el conjunto de circunstancias que llevaron a concebir esta función.
- `mbstring`
    - Se trata de "multi-byte", cuando el problema es realmente el charset.
    - Funciona en cadenas normales. Tiene un único charset global "por defecto". Algunas funciones permiten especificar charset, pero a partir de ahí se aplica a todos los argumentos y sus valores de retorno.
    - Provee funciones `ereg_*` pero están deprecadas. `preg_*` no sirven, aunque pueden entender el UTF-8 si se especifica una bandera PCRE específica.
- PHP puede supuestamente parsear XML. No puede en realidad. Puede encontrar correspondencias de un patrón en archivos de texto con una estructura apropiada, escritos con ciertos conjuntos de caracteres utilizados en una parte del mundo.

  Para complicar el problema muchos desarrolladores, en especial novatos en PHP, no tienen absolutamente ninguna idea de lo que es un conjunto de caracteres, así que el problema es difícil de explicar. Para complicar más aun, MySQL, la base de datos más usada en conjunción con PHP, tiene un soporte de conjunto de caracteres bastante malo hasta la versión 5.0, y aunque ha mejorado mucho desde entonces, todavía es notablemente malo. (No usa UTF-8 para la codificación por defecto, por ejemplo).

  Interesantemente, las personas que usan alfabetos occidentales nunca se encuentran con este problema hasta que su aplicación se convierte en popular y alguien en Rusia trata de dejar un comentario en ruso, y su foro falla de repente. Algunos otros, como los desarrolladores de ezPublish, vienen de Noruega y reconocen el problema de PHP desde el comienzo mismo.

- Sin soporte Unicode. Solo ASCII trabaja realmente bien. Existe la extensión `mbstring`, pero es bastante mala. Sobre esto incluso Joel Spolsky tuvo algo que decir en su momento:

  "Cuando descubrí que la popular herramienta de desarrollo web, PHP, tiene una ignorancia casi completa de los problemas derivados de los conjuntos de caracteres, usando descaradamente 8 bits para los caracteres y por tanto haciendo casi imposible desarrollar aplicaciones con buena internacionalización, pensé ya es suficiente.

> *Así que tengo un anuncio que hacer: Si es un programador en el 2003* [o con más razón, en el 2012] *y no conoce lo básico de conjuntos de caracteres, codificaciones y Unicode, lo voy a encontrar, y a castigar haciéndolo pelar cebollas por seis meses dentro de un submarino. Estoy hablando en serio, lo haré. Y algo más:* ***NO ES TAN DIFICIL****."*
>
> Y esto fue en el 2003. Años después, no hay aún soporte Unicode.

- Utilizar las funciones del lenguaje en cadenas UTF-8 es correr el riesgo de corromperlas.
- Similarmente, no existe el concepto de comparación de mayúsculas y minúsculas fuera del ASCII. A pesar de la proliferación de versiones sin sensibilidad de funciones, ninguna de ellas considera `é` igual a `É`.
- No se pueden utilizar comillas cuando se interpolan variables, por ejemplo, `"$foo['key']"` es un error de sintaxis. Se pueden quitar las comillas (lo cual podría generar una advertencia en otro lugar) o usar `${...}`/`{$...}`.
- `"${foo[0]}"` está bien. `"${foo[0][0]}"` es un error de sintaxis. Poner el `$` dentro funciona en ambas versiones. Esto es una mala copia de una sintaxis semejante de Perl (con semánticas radicalmente diferentes)?

Se me ha mencionado que PHP 6 finalmente introducirá soporte Unicode. Lo cual va a ser otro cambio incompatible más. Supongo que veremos de nuevo `<sarcasm>`otra adopción en gran escala como la de PHP 5`</sarcasm>`.

Sistema y reflection

- Existen, en general, un grupo grande de funciones que difuminan la diferencia entre texto y variables. De estas `compact` y `extract` son solo la punta del iceberg.
- Hay varias formas de ser dinámico en PHP, y a primera vista no existen diferencias obvias o beneficios relativos. `classkit` puede modificar las clases definidas por los usuarios; `runkit` lo reemplaza, y puede modificar cualquier cosa definida por el usuario; las clases `Reflection*` pueden funcionar en la mayor parte del lenguaje; además hay una gran cantidad de funciones individuales para reportar propiedades de funciones y clases. ¿Estos sistemas son independientes, relacionados, o redundantes?

`php_uname` da información sobre el SO actual. A no ser que PHP no pueda definir en qué sistema se está ejecutando, entonces te da información sobre el SO en que fue compilado. En ningún caso te dice si esta situación se presentó.

- `get_class($obj)` retorna el nombre de clase del objeto. `get_class()` retorna el nombre de la función en que se está llamando. Dejando a un lado que estas dos funciones hacen cosas radicalmente diferentes: `get_class(null)` … actúa como la segunda. Así que no se puede confiar en ella con un valor arbitrario en `$obj`. ¡Sorpresa!

- Las clases `stream_*` permiten la implementación de objetos de flujo personalizados para usar con `fopen` y otras funciones del sistema para archivos. "tell" no puede ser implementada por razones internas.
- `register_tick_function` acepta un objeto closure. `unregister_tick_function` no lo hace, en vez de eso lanza un error quejándose de que el objeto no se puede convertir en cadena.
- `php_uname` da información sobre el SO actual. A no ser que PHP no pueda definir en qué sistema se está ejecutando, entonces te da información sobre el SO en que fue compilado. En ningún caso te dice si esta situación se presentó.
- `fork` y `exec` no son construcciones del sistema. Vienen con la extensión `pcntl` , pero no se incluyen por defecto. `popen` no proporciona un pid.
- El valor de retorno de stat's está cacheado.
- `session_decode` existe para leer una cadena arbitraria de sesión de PHP, pero solo funciona si ya existe una sesión activa. Y vuelca el resultado a `$_SESSION` en vez de retornarlo.
- El manejo de sesiones de PHP tampoco es multihilo. Por defecto PHP usa sesiones basadas en disco, o sea en archivos, utilizando bloqueos en el fichero tan pronto como se use `session_start`(). Esencialmente esto asegura que varias peticiones de la misma sesión solo funcionen secuencialmente, una tras otra. Puede aliviarse este problema cerrando las sesiones con `session_write_close()` o usando una solución como Sharedance.

Miscelánea

- `curl_multi_exec` no cambia `curl_errno` bajo un error, pero si cambia `curl_error`.
- Los argumentos de `mktime` son, en orden: hora, minuto, segundo, mes, día, año.
- Algunas funciones (como `money_format()`) solo están definidas en algunas plataformas y no en otras. Nótese que `money_format()` no es una función de una extensión, es parte de la librería estándar!
- Formatos de fecha no estándar: Muchos programadores están familiarizados con los formatos de fecha de Unix y C. Otros lenguajes lo adoptaron, pero PHP tiene el suyo propio e incompatible. Mientras en C %j es el día del año, en PHP es el día del mes. Para confundir más aun, `strftime` y `date_format` de Smarty usan los formatos C/UNIX.
- Incrementar (`++`) un `NULL` produce `1`. Decrementar (`--`) un `NULL` produce `NULL`. Decrementar una cadena también la deja sin cambios.

Los programas no son más que grandes máquinas que mastican datos y escupen otros datos. Muchos lenguajes están diseñados alrededor de los datos que manipulan, desde awk hasta Prolog y C. Si un lenguaje no puede manejar datos, no puede hacer nada.

Números

- Los enteros son con signo y de 32-bit en plataformas de 32-bit. A diferencia de los contemporáneos de PHP, no hay promoción automática de bigint. Así que se pueden terminar con sorpresas como tamaños negativos para archivos, y la matemática puede variar en dependencia de la *arquitectura del CPU*. La única opción para enteros grandes es utilizar las funciones wrapper GMP o BC. (Los desarrolladores han propuesto adicionar un tipo nuevo, separado, de entero de 64 bits. Esto es una locura.)

- El soporte de PHP para enteros grandes es exactamente tan idiota como su manejo de Unicode y su sistema de tipos.

  Los enteros en PHP son de 32 bits con signo en plataformas de 32 bits, y de 64 con signos en plataformas de 64 bits. No hay forma oficial de determinar el tamaño de un entero. Teóricamente, si se fuera a sacar un entero de 32 sin signo de un flujo binario, se usaría `unpack()`. Pero esto convertiría un número demasiado grande en un float si no cabe en un entero de 32, a no ser que se esté ejecutando PHP 5.2.1, porque esa versión simplemente explotaba en el `unpack()`.

  Así que si quieren escribir un programa en PHP que maneje enteros mayores que 2^31 en cualquier plataforma, no pueden usar los operadores normales (+ - * /), sino que necesitan usar la librería `bcmath`. Y no pueden usar `unpack()`, debido a una de mis notas favoritas en la documentación de PHP:

  "Nótese que PHP internamente guarda valores enteros como valores con signo. Si se desempaca un número muy grande sin signo y es del mismo tamaño que los valores guardados internamente por PHP, el resultado será un número negativo, incluso si se especifica el desempaque como sin signo."

  ¿Quieren un entero sin signo? No, lo siento, no pueden pedirnos eso. ¡Espero que no estén escribiendo una aplicación para contar votos en elecciones! En un blog de MySQL encontraron una función portable para resolver este problema, que se ve así:

```
function _Make64 ( $hi, $lo ) {
   // on x64, we can just use int
   if ( ((int)4294967296)!=0 )
     return (((int)$hi)<<32) + ((int)$lo);

   // workaround signed/unsigned braindamage on x32
   $hi = sprintf ( "%u", $hi );
   $lo = sprintf ( "%u", $lo );

   // use GMP or bcmath if possible
   if ( function_exists("gmp_mul") )
     return gmp_strval ( gmp_add ( gmp_mul ( $hi, "4294967296" ), $lo )
);
   if ( function_exists("bcmul") )
    return bcadd ( bcmul ( $hi, "4294967296" ), $lo );

   // compute everything manually
   substr ( $hi, 0, -5 );
   $b = substr ( $hi, -5 );
   $ac = $a*42949; // hope that float precision is enough
   $bd = $b*67296;
   $adbc = $a*67296+$b*42949;
   $r4 = substr ( $bd, -5 ) +  + substr ( $lo, -5 );
   $r3 = substr ( $bd, 0, -5 ) + substr ( $adbc, -5 ) + substr ( $lo,
0, -5 );
   $r2 = substr ( $adbc, 0, -5 ) + substr ( $ac, -5 );
   $r1 = substr ( $ac, 0, -5 );
   while ( $r4>100000 ) { $r4-=100000; $r3++; }
```

```
    while ( $r3>100000 ) { $r3-=100000; $r2++; }
    while ( $r2>100000 ) { $r2-=100000; $r1++; }

    $r = sprintf ( "%d%05d%05d%05d", $r1, $r2, $r3, $r4 );
    $l = strlen($r);

    $i = 0;
   while ( $r[$i]=="0" && $i<$l-1 )
      $i++;

    return substr ( $r, $i );
}
```

Este es el tipo de código que se debe escribir en PHP para lograr que sea portable y predecible. ¿Quieren volver a C++? No los culpo.

- PHP soporta números octales con un cero a la izquierda, por ejemplo, `012` es el número diez. Sin embargo `08` se convierte en cero. El 8 (o 9) y cualquier dígito que los siga desaparece. `01c` es un error de sintaxis.

```
/tmp$ php -r 'print 07'

Parse error: parse error in Command line code on line 1

/tmp$ php -r 'print 07;'
7

/tmp$ php -r 'print 08;'
0
```

Anjá. Como lo ven.

Una cosa menos tonta hubiera sido dar una advertencia al usuario, o fallar. Simplemente dar el número equivocado es peligroso, porque el usuario puede pensar que su código está bien después de ver un par de ejemplos:

```
$magic_dates = array(stamp(01,01,1980),
stamp(12,31,1980),
stamp(08,11,1950)); # fails silently!
```

PHP no dice que nada está mal, incluso con su nivel más pedante de advertencia:

```
error_reporting(E_ALL); eval("print 028;");  # prints 2 without warning
```

¿Por qué? ¿Pereza? ¿Ignorancia? ¿Ambas?

- `0x0+2` produce 4. El parser considera el `2` como parte de la literal hexadecimal *y* como un decimal separado, tratando esto como `0x002 + 2`. `0x0+0x2` produce el mismo problema. Extrañamente, `0x0 +2` también es 4, pero `0x0+ 2` se calcula correctamente como 2. (Esto está

arreglado en PHP 5.4. Pero también se volvió a romper en el mismo PHP 5.4, con el nuevo prefijo `0b`: `0b0+1` también produce 2.)

- Adivinen que hace este programa

```
error_reporting(E_ALL);  # inútil :)
foreach(array(0xffffff,
              0xfffffff,
              0xffffffff,
              0xfffffffff,
              0xffffffffff) as $x) {
  printf("%s (%s)\n", $x, gettype($x));
}
```

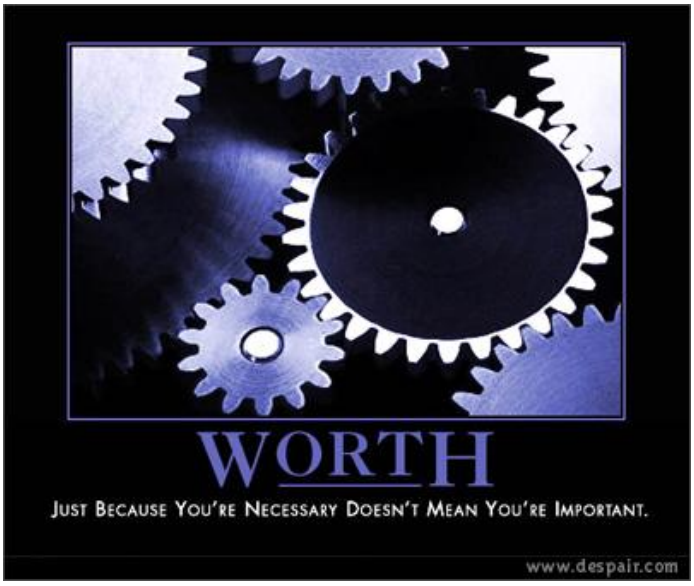
Salida:

```
16777215 (integer)
268435455 (integer)
4294967295 (double)
2147483647 (integer)
2147483647 (integer)
```

- `pi` es una función. O se puede usar la constante `M_PI`.
- No hay operador de exponenciación, solo la función `pow`.

Arreglos

- Este único tipo de datos actúa como una lista, mapa hash ordenado, conjunto ordenado, lista dispersa, y ocasionalmente alguna rara combinación de los anteriores. ¿Cuál es su rendimiento? ¿Cómo lo hace? ¿Qué tipo de uso de memoria tenemos aquí? ¿Quién lo sabe? De todas maneras, no es como si hubiera ninguna otra opción.
- `=>` no es un operador. Es una construcción especial que existe solo dentro de `array(...)` y `foreach`.



Este único tipo de datos actúa como una lista, mapa hash ordenado, conjunto ordenado, lista dispersa, y ocasionalmente alguna rara combinación de los anteriores (…) De todas maneras, no es como si hubiera ninguna otra opción.

- Los índices negativos no funcionan, dado que `-1` es una llave tan válida como `0`.
- A pesar de que esta es la única estructura de datos del lenguaje, no tiene sintaxis corta; `array(...)` *es* la sintaxis corta. (PHP 5.4 trae "literales", `[...]`.)
- `=>` es basada en Perl, que permite hacer `foo => 1` sin comillas. (De hecho, esta es la razón por la cual existe en Perl.) En PHP, no se puede hacer esto sin que se produzca una advertencia; es el único lenguaje de su tipo que no tiene manera de crear un hash sin ponerle comillas a una cadena.
- Las funciones de arreglo tienen comportamiento inconsistente o confuso, porque operan sobre listas, hashes, o una combinación de ambos. Consideren `array_diff`, que "calcula la diferencia entre dos arreglos".

```
$first  = array("foo" => 123, "bar" => 456);
$second = array("foo" => 456, "bar" => 123);
echo var_dump(array_diff($first, $second));
```

¿Qué hace este código? Bueno, si `array_diff` trata sus argumentos como hashes, entonces obviamente son diferentes, las mismas llaves tienen valores distintos. Si los trata como listas, también son diferentes, los valores no están en el mismo orden.

En realidad `array_diff` los considera igual, porque los trata como conjuntos: compara solo los valores, e ignora el orden.

- Similarmente, `array_rand` tiene el comportamiento extraño de seleccionar llaves aleatorias, lo cual no es de mucha ayuda para el caso más común en el que se necesita seleccionar de una lista de opciones.
- A pesar de lo mucho que PHP depende de preservar el orden de las llaves:

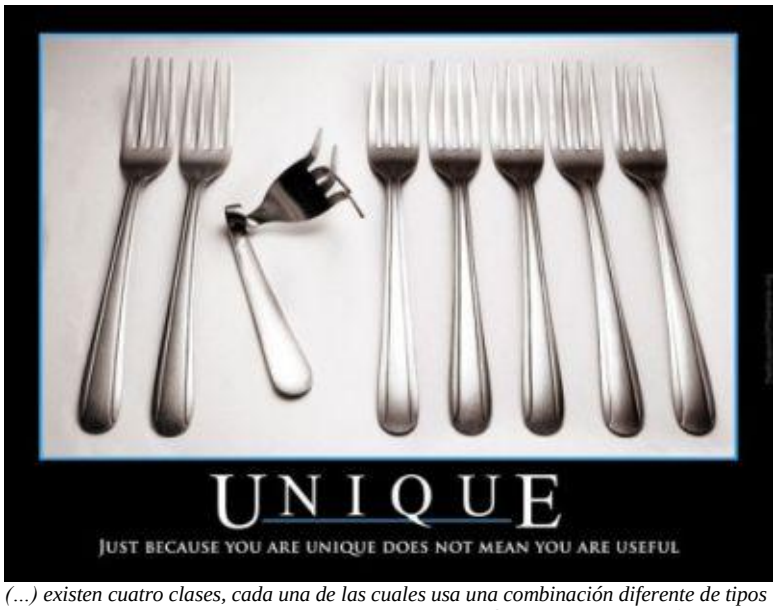
```
array("foo", "bar") != array("bar", "foo")
array("foo" => 1, "bar" => 2) == array("bar" => 2, "foo" => 1)
```

Les dejo de tarea averiguar qué pasa cuando se mezclan los arreglos. Yo no tengo idea.

- `array_fill` no puede crear arreglos de longitud cero; en vez de ello lanza una advertencia y regresa falso.
- Todas de las muchísimas funciones de ordenamiento operan en memoria y no retornan nada. No hay manera de crear una copia nueva ordenada, tienen que primero copiar el arreglo y luego ordenarlo manualmente.
- Pero `array_reverse` sí retorna un arreglo nuevo.
- Una lista de cosas ordenadas y un poco de mapeo de llaves a valores parece una gran cosa para manejar parámetros de función, pero no,

No Arreglos

- La librería estándar incluye “Quickhash”, una implementación OOP de “clases específicamente fuertemente tipadas” para implementar hashes. Y seguro, existen cuatro clases, cada una de las cuales usa una combinación diferente de tipos de llave y valor. No queda claro porque la implementación de arreglos por defecto no puede optimizar estos casos tan extremadamente comunes, o cual es la ganancia relativa de rendimiento.
- Existe una clase `ArrayObject` (que implementa *cinco* diferentes interfaces) que puede envolver un arreglo y hacer que actúe como un objeto. Las clases definidas por el usuario también pueden implementar estas interfaces. Pero solo tiene un puñado de métodos, la mitad de los cuales no se parecen a las funciones definidas por defecto, y las funciones de arreglo definidas por defecto no saben cómo operar en un `ArrayObject` o cualquier otra clase que parezca un arreglo.



(...) existen cuatro clases, cada una de las cuales usa una combinación diferente de tipos de llave y valor. No queda claro porque la implementación de arreglos por defecto no puede optimizar estos casos tan extremadamente comunes, o cual es la ganancia relativa de rendimiento.

Funciones

- Las funciones no son datos. Las closures son objetos, pero las funciones regulares no. No pueden siquiera ser llamadas por sus nombres: `var_dump(strstr)` lanza una advertencia y asume que se quiso usar la cadena literal `"strstr"`. No hay manera de distinguir entre una cadena arbitraria o una “referencia” a función.
- `create_function` es básicamente un wrapper de `eval`. Crea una función con un nombre normal y lo instala globalmente (nunca va a ser recogido por el recolector de basura – no lo usen en un ciclo!) No tiene conocimiento ninguno sobre el contexto donde está. El nombre siempre contiene un byte con valor NUL para que nunca tenga conflictos con una función regular.
- Declarar una función nombrada `__lambda_func` rompe el `create_function`. Si `__lambda_func` ya existe, esto lanza un error fatal.

Ejecución

- Un solo archivo compartido, `php.ini`, controla partes masivas de las funcionalidades de PHP e introduce reglas complejas acerca de qué cosa sobrescriba que cosa y cuando lo hace. El software de PHP que se espera desplegar en máquinas arbitrarias tiene que sobrescribir muchas

configuraciones para normalizar su ambiente, lo cual hace que un mecanismo como `php.ini` no sea útil en muchas ocasiones.
    - PHP busca el `php.ini` en una variedad de lugares, así que puede (o no) ser posible sobrescribirlo en un servidor. En algunas plataformas (Windows con Apache 2.2.x y PHP 5.3, por ejemplo) solo **un** archivo de este tipo puede ser parseado, así que no se puede sobrescribir solo un par de configuraciones y listo, hay que sustituir todo el archivo en cada caso. Y se asume que el programador tiene control sobre el servidor al punto de modificar este archivo, lo cual no siempre es cierto.
- PHP básicamente es un CGI. Cada vez que se solicita una página PHP la recompila antes de ejecutarla. Incluso los servidores de desarrollo para Python (o Java) no hacen esto.

  Esto ha propiciado todo un Mercado para aceleradores de PHP, que solo compilan una vez, acelerando PHP a la velocidad de cualquier otro lenguaje. Zend ha convertido esto en parte de su modelo de negocio, lo cual lleva a pensar que PHP es lento por razones comerciales. La velocidad de PHP se puede aumentar en un 500% usando cache. Y si es así, porque el cache no es una funcionalidad de PHP? Porque Zend, creador de PHP, vende su propio Zend Accelerator, y por supuesto no quieren dañar sus propios productos comerciales. La alternativa es APC.

- ¿Creen que Java es lento? ¡PHP lo es mucho más! Vean la comparación en Computer Language Shootout. Entonces, como es que PHP puede usarse en sitios populares con muchísimos visitantes? Debido al cache. Estos sitios usan memcached o APC para mejorar el rendimiento. No prueba nada sobre PHP, excepto que es cacheable. Incluso Rasmus Lerdorf, el creador de PHP lo admite, en una entrevista de SitePoint, donde le preguntan cómo hacer que PHP sea más rápido. Su respuesta: “no se puede”.
- Shared Nothing

  Un intérprete lento no sería un gran problema si se pudiera resolver adicionando un par de servidores extra. Pero en PHP, que no tiene soporte multihilo, no se pueden compartir datos entre distintos procesos. Los partidarios de PHP llaman a esto la arquitectura “Share Nothing”. Muy inteligente vender un problema como una ventaja. En cualquier sistema se deben compartir datos, y cuando hay que hacerlo en PHP, la solución común es mover el problema a la (ya bastante cargada) base de datos. ¿Es esta la solución correcta? No me parece.

  Un script simple de PHP puede pesar menos de 100kB. Pero una página en un sistema serio o en un CMS puede fácilmente alcanzar 15 MB (sí: megabytes). Para cada petición esta memoria tiene que ser inicializada, lo que lleva tiempo y limita las conexiones concurrentes. Si tenemos 2 GB disponibles, solo se pueden responder 2000/15=133 conexiones simultáneas. No mucho, en realidad.

  En la misma entrevista mencionada anteriormente, Rasmus admite que PHP no escala bien.

  Facebook es un buen ejemplo para probar esto. Facebook es probablemente el sitio de PHP más grande del mundo. En el 2004, tenían 2.3 millardos de visitas al mes. También usaban 30 mil servidores. Eso es mucho. Si se hacen algunos cálculos, encontraremos que, en promedio, un servidor responde 106 visitas por hora. Eso es realmente un rendimiento, muy malo. Y Facebook no es un sitio de procesamiento tan pesado, es texto y fotos mayormente.

- Durante mucho tiempo, los errores de PHP iban al cliente por defecto – Supongo que para ayudar durante el desarrollo. No creo que este sea el caso todavía, pero todavía aparecen uno que otro error de MySQL en el tope de la página. Los usuarios de Drupal me apoyarán en esto.
- PHP está lleno de extraños "easter eggs", como producir el logo del PHP si se pasa el argumento correcto a la URL. No solo es esto completamente irrelevante a la construcción de la aplicación, sino que permite detectar la tecnología (y quizás hasta la versión) sin importar cuanto `mod_rewrite`, FastCGI, reverse proxying, o configuraciones `Server:` hayas hecho. (corregido en PHP 5.5 alpha)
- Las líneas en blanco antes o después de las etiquetas `<?php ... ?>`, incluso en las librerías, cuentan como texto y son enviadas a la respuesta (o causa "headers already sent"). Las opciones son evitar estrictamente las líneas en blanco al final de cada archivo o no poner la etiqueta `?>`.

Despliegue

El despliegue se cita con frecuencia como la mayor ventaja de PHP: Copiar algunos ficheros y listo. Esto es ciertamente mucho más sencillo que ejecutar todo un proceso como Python, Ruby o Perl. Pero PHP deja mucho que desear.

En general, estoy a favor de separar el servidor de aplicación del servidor web. Toma un esfuerzo mínimo configurar esto, y los beneficios son muchos: se pueden administrar los servidores por separado, se pueden correr varios procesos sin necesidad de más servidores web, se puede ejecutar la aplicación con otro usuario, cambiar el servidor web, desplegar cambiando la configuración de la lista de peticiones, etc. Casar las aplicaciones a un servidor web es absurdo y no hay una buena razón para hacerlo.

- PHP está naturalmente atado a Apache. Utilizarlo por separado, o con cualquier otro servidor web requiere tanta configuración (o más) que cualquier otro lenguaje.
- En Windows, al menos hasta la versión 5.3.9 y el Apache 2.2.21 sigue sin ser posible especificar dos `php.ini` distintos. En Linux, sin embargo, lo es, lo cual me lleva a pensar que este es un efecto controlable mediante una bandera de compilación. Por qué este comportamiento no es por defecto (que debería serlo, por su utilidad), no tengo idea.
- Además de ello, montar una aplicación PHP sin tener en cuenta las configuraciones de `php.ini` del servidor que provee el hosting (que puede muy bien no estar bajo el control del administrador de la aplicación) es riesgoso, por lo cual la aplicación necesita normalizar su ambiente con múltiples llamadas a `ini_set`, o utilizando `.htaccess`. Ahí termina la tan famosa "cero fricción" y la portabilidad de aplicaciones de PHP entre dos servidores Apache distintos. Y no hablemos de lo que pasa si son sistemas operativos o versiones de Apache distintas…
- Similarmente, no hay una manera sencilla de aislar una aplicación PHP y sus dependencias del resto de un sistema. Correr dos aplicaciones que requieren versiones diferentes de una librería, o del mismo PHP? Comencemos por montar otro Apache…
- El soporte de PHP es inconsistente entre plataformas. Muchas cosas posibles en Linux no lo son en Windows, y frecuentemente las versiones del lenguaje (y de sus herramientas relacionadas, principalmente Apache) son compiladas con banderas distintas de la distribución Linux estándar. Esto atenta contra uno de los puntos fuertes del lenguaje, la ubicuidad de su entorno.
- El enfoque del "montón de archivos", además de hacer que rutear sea un gran problema, también hace que haya que cuidadosamente autorizar o denegar acceso a todos los recursos del

sitio, porque la jerarquía de las URL es también el árbol completo de código. Los archivos de configuración y otros necesitan protecciones al estilo C para que no se puedan cargar directamente. Los archivos de control de versiones (ej., `.svn`) necesitan protección. Con `mod_php`, *todo* en el sistema de archivos es un potencial punto de entrada. Con un servidor de aplicaciones, hay un solo punto de entrada, y solo la URL controla si se invoca o no.

- No se puede actualizar sin interrupciones un grupo de ficheros que corren al estilo CGI, a no ser que se quieran explotes y comportamiento indefinido si los usuarios solicitan algo del sitio a medio camino de la actualización.

- A pesar de lo "simple" que es configurar Apache con PHP, existen algunas trampas sutiles incluso aquí. Mientras los documentos de PHP sugieren usar `SetHandler` para hacer que los archivos .php se ejecuten como PHP, `AddHandler` parece funcionar igual de bien, y de hecho en Google aparecen casi el doble de resultados con el segundo que con el primero. Este es el problema:

  Al usar `AddHandler`, estamos diciéndole a Apache que "ejecutar esto como php" es una de las formas posibles de manejar los archivos `.php`. ¡**Pero**! Apache no tiene la misma idea de lo que es una extensión de fichero que cualquier ser humano en el planeta. Está diseñado para soportar, por ejemplo, `index.html.en` y reconocerlo como inglés y HTML. Para Apache, un fichero puede tener cualquier número de extensiones simultáneamente.

Para asegurarse de que nadie suba código PHP, chequeamos que el archivo no tenga extensión php. Todo lo que tiene que hacer el atacante es subir un archivo foo.php.txt. El formulario no ve ningún problema, pero Apache reconoce el archivo como PHP, y lo ejecuta alegremente.

  Ahora imaginemos que tienen un formulario para subir ficheros y dejarlos en un directorio público. Para asegurarse de que nadie suba código PHP, chequeamos que el archivo no tenga extensión php. Todo lo que tiene que hacer el atacante es subir un archivo `foo.php.txt`. El formulario no ve ningún problema, pero Apache *reconoce* el archivo como PHP, y lo ejecuta alegremente.

  El problema aquí no es "usar el nombre original del fichero" o "validar mejor". El problema es que el servidor web está configurado para correr cualquier código viejo con el que se encuentre, que es precisamente la misma propiedad que hace que PHP sea "fácil de desplegar". CGI requiere `+x`, que por lo menos era algo, pero PHP ni siquiera hace eso. Y este no es un problema teórico, existen varios sitios en producción con este problema.

Funcionalidades faltantes

Considero que todas estas son funcionalidades más o menos críticas para construir una aplicación web. Parecería razonable que PHP, dado que su mayor característica es ser un lenguaje Web, debería tener alguna de ellas.

- Sin sistema de plantillas. Está PHP en sí mismo, pero nada que actúe como un intermediario en vez de un programa.
- Sin filtro XSS. No, "recordar usar `htmlspecialchars`" no es un filtro XSS.
- Sin protección CSRF. Tienes que hacerlo tú mismo.
- Sin API genérica estándar para bases de datos. Cosas como PDO tienen que utilizar cada una de las APIs individuales para poder abstraer las diferencias.
- Sin ruteo. Un sitio Web se ve exactamente como el sistema de archivos donde está. Muchos desarrolladores han sido engañados para que piensen que `mod_rewrite` (y `.htaccess` en general) es un sustituto aceptable.
- Sin autenticación ni autorización.
- Sin depuración interactiva (la extension X-Debug es separa del lenguaje y es relativamente compleja de instalar e integrar con los IDEs).
- Sin un mecanismo coherente de despliegue, solo "copia estos archivos al servidor".
- El API de validación de la función `filter_var` recibe filtros para varios valores útiles, excepto para date (no existe el `FILTER_VALIDATE_DATE`, aunque si existe `FILTER_VALIDATE_EMAIL` y `FILTER_VALIDATE_IP`).

Fronteras del lenguaje

La poca reputación de seguridad de PHP se debe a que toma datos arbitrarios de un lenguaje y los lanzan en otro lenguaje. Esto es una mala idea. `<script>` no significa nada en SQL, pero sí que significa algo en HTML.

Empeorando esta situación está la llamada común a "sanear las entradas". Esto está totalmente *mal*, no se puede sacudir una varita mágica y hacer que un pedazo de data sea inherentemente limpio. Lo que es necesario es conocer el lenguaje: usar placeholders con SQL, usar listas de argumentos para lanzar procesos, etc.

- PHP estimula directamente el "saneamiento": Hay toda una extensión de filtrado de datos para hacerlo.
- Todas las funciones `addslashes`, `stripslashes`, y otras tonterías con slashes son soluciones falsas para despistar.
- No hay forma, hasta donde sé, de lanzar un proceso de manera segura. Solo se puede ejecutar una cadena a través de una consola, así que las opciones son escapar la cadena como un maniático (y confiar que la consola que usas tenga el mismo tipo de escape) o usar `pcntl_fork` y `pcntl_exec` *manualmente*.
- Tanto `escapeshellcmd` como `escapeshellarg` tienen más o menos la misma descripción. Nótese que en Windows, `escapeshellarg` no funciona (porque asume que está trabajando con una consola Bourne), y `escapeshellcmd` solo reemplaza un grupo de signos de puntuación con espacios porque nadie puede trabajar con el escape del cmd de Windows (que silenciosamente rompe lo que sea que estén tratando de hacer).

- Las funciones originales de MySQL, que todavía están en uso generalizado, no tienen manera de crear llamadas preparadas.
- Algunas funciones críticas de seguridad de PHP se rompen si se usan con texto en algunos conjuntos de caracteres particulares. Existe una lista de conjuntos de caracteres soportados.

Hasta hoy, la documentación de PHP sobre inyección de SQL recomienda prácticas raras como chequeo de tipos, usar `sprintf` e `is_numeric`, usar `mysql_real_escape_string` manualmente en todas partes, o usar `addslashes` manualmente por todas partes (que "puede ser útil"). No existe ninguna mención de PDO o parametrización, excepto en los comentarios de los usuarios.

Inseguro por defecto

- `register_globals`. Ha estado desactivado por defecto por un tiempo, y fue eliminado en 5.4. No importa, esto es una *vergüenza*.
- `include` que acepta HTTP URLs. Ver punto anterior.
- Comillas mágicas. Tan cerca de ser seguro por defecto, y sin embargo tan lejos de entender el concepto en absoluto. Y, ver puntos anteriores.

Núcleo

El intérprete de PHP tiene algunos problemas de seguridad propios que resultan *fascinantes*.

- En 2007 este intérprete tenía una vulnerabilidad de desbordamiento de enteros. El arreglo empezó con `if (size > INT_MAX) return NULL;` y se fue poniendo peor. (Para los que no saben C: `INT_MAX` es el mayor valor posible para una variable en cualquier circunstancia. Espero que vean el problema con esto).
- Más recientemente, PHP 5.3.7 se las ingenió para incluir una función `crypt()` que efectivamente deja a cualquiera autenticarse sin contraseña.
- El servidor de desarrollo de PHP 5.4 es vulnerable a un ataque DOS, debido a que toma el encabezado `Content-Length` (que cualquiera puede configurar a cualquier tamaño) y trata de reservar esa cantidad de memoria. Esta es una mala idea.

*El intérprete de PHP tiene algunos problemas de seguridad propios que resultan **fascinantes***

Podría desenterrar otros problemas, pero el punto no es decir que existen X exploits – EL software tiene errores, estas cosas pasan. Es la naturaleza de estos errores lo que es aterrorizante.

Y ni siquiera estaba buscando estos que listo. Simplemente aparecieron en mi trabajo normal durante los últimos meses.

Conclusiones

Mantengo que de todos los lenguajes de tipado débil PHP tiene la sintaxis más aceptable, especialmente para desarrolladores de los lenguajes fuertemente tipados. Supongo que esto influyó e influye en la gran popularidad del lenguaje en este momento. También, la gran disponibilidad de aplicaciones sofisticadas y pulidas es importante. Desde mi punto de vista parecen existir muchos más sistemas de manejo de contenido y otras herramientas escritas en PHP que en Python o en Ruby.

PHP tiene dos ventajas: es fácil de aprender y está soportado por muchas compañías de hospedaje web. Pero eso no lo hace un buen lenguaje.

Una vez reconocido esto, también reconozco que es muy importante elegir la herramienta adecuada para el trabajo que se esté realizando. Y por supuesto, esto significa que PHP debe elegirse en las circunstancias adecuadas:

- El software ideal para basar tu aplicación está escrito en PHP
- Ya existe una gran inversión en la tecnología
- Existen restricciones de tiempo que no permiten aprender otra tecnología.

Para mí, PHP es una solución inaceptable excepto para los problemas más simples. Por lo menos si han leído este artículo completo ya conocen muchos de los problemas principales y algunas maneras de evitarlos. Pero no piensen que eso es suficiente para escribir fácilmente código estable, mantenible y predecible en PHP, incluso usando PHP 5 y un framework como Symfony. En mi opinión desarrollar en PHP es tan difícil como desarrollar en C, excepto que es más peligroso porque no es aparente la dificultad hasta que es demasiado tarde.

Es importante que lean los artículos listados en las referencias. Contienen mucha información adicional, sobre todo comentarios a favor y en contra de cada uno de los argumentos dados. Tengan en cuenta que los artículos tomados como fuente son de fechas diversas y no todos los debates están activos aún (asegúrense también de que el problema sea válido en la versión de PHP que utilicen).

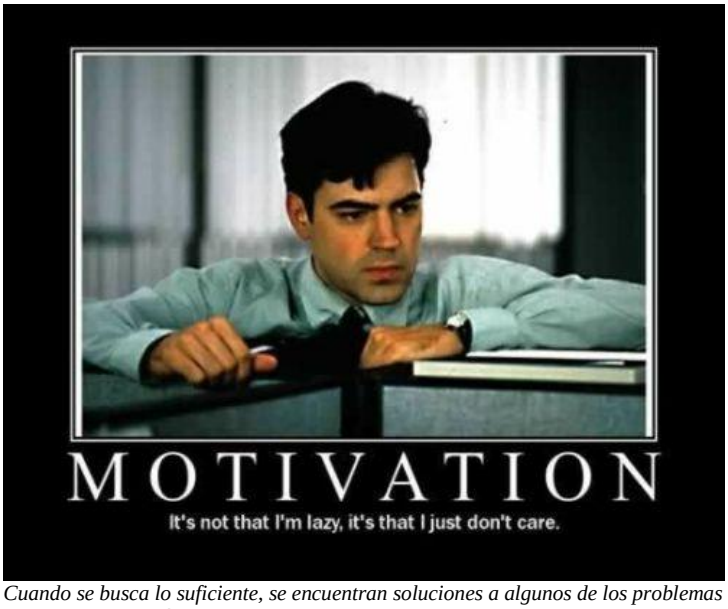
Si ya existe una inversión en PHP, hay maneras de hacer que funcione. Ninguno de los problemas mencionados en este artículo es sin solución o parche. El problema es hacer que la solución sea mantenible, predecible y que esté bien documentado en la plataforma utilizada.

Usar frameworks es una necesidad, no una opción. Symfony, Doctrine y semejantes ayudan a aislar estos problemas del lenguaje en muchos casos, y pueden hacer a los desarrolladores más productivos a pesar de su elección de tecnología poco afortunada. Si un framework es demasiado pesado, consideren al menos utilizar un motor de plantillas como Smarty o Twig. La ganancia en velocidad y mantenimiento ciertamente lo vale.

Espero que los haya hecho pensar un poco. El objetivo no es demostrar que Java, C#, Python o Ruby no tienen problemas, sino que PHP tiene realmente **muchos, muchísimos**☺. Conocer las limitaciones del lenguaje puede ayudarnos a ser mejores programadores.

Para proyectos muy pequeños, PHP puede ser un buen lenguaje de programación. Para proyectos más grandes y complicados, PHP comienza a mostrar sus debilidades. Cuando se busca lo suficiente, se encuentran soluciones a algunos de los problemas mencionados. ¿Y sí se conocen estas soluciones, por qué no se arreglan? ¿Y por qué las soluciones no se mencionan en el manual?

Es bueno que un lenguaje de código abierto sea tan popular. Desafortunadamente, no es un buen lenguaje. Espero que en algún momento todos los problemas sean resueltos (¿en PHP 6, quizás?) y que tengamos un lenguaje que sea código abierto **y bueno**.



Cuando se busca lo suficiente, se encuentran soluciones a algunos de los problemas mencionados. ¿Y sí se conocen estas soluciones, porque no se arreglan?

Hasta entonces, cuando vayan a comenzar un proyecto de más de 5 páginas, consideren utilizar C#/ASP.Net o Java/JSP como una mejor solución. Si tiene absolutamente que ser completamente libre, existen Python y Ruby. Qt le ha dado nueva fuerza al C++ como lenguaje, y otros ecosistemas han surgido mientras los establecidos luchan entre sí, como Scala sobre la JVM. Y si realmente quieren buen rendimiento, pueden estudiar algunas alternativas que han estado creciendo fuerte en los últimos años: Nginx, Couch DB o Node.js, por mencionar solo algunas.

Inviertan en estudiar diseño orientado a objetos, patrones y los frameworks de PHP más actuales. Este trabajo disminuirá muchísimo el impacto de las características del lenguaje seleccionado en la aplicación que están construyendo, aislándolos de los problemas del lenguaje detrás de las soluciones de diseño y las herramientas. Es una lástima que sea una gran inversión en capacitación, pero es necesaria y vale la pena.

En resumen:

- Si son programadores y comienzan un proyecto nuevo, NO elijan PHP. Se arrepentirán más tarde o más temprano.
- Si son programadores y tienen que continuar un proyecto de PHP ya existente, reescríbanlo en un lenguaje más sano.
- Si no pueden reescribirlo, utilicen este documento para buscar y corregir tantos problemas como sea posible, utilicen Frameworks y documenten cada decisión que tomen. Les será útil.
- Si son estudiantes buscando un nuevo lenguaje, lean un libro de C#, Java o Python. Son mucho más productivos e intelectualmente estimulantes, y contribuirán igualmente a sus capacidades profesionales.
- Si son estudiantes buscando tecnología para una tesis de grado, maestría, hobby o trabajo final, PHP puede servirles siempre y cuando determinen a priori que su trabajo no tiene ni va a tener uso real donde sea necesario darle mantenimiento ni continuidad, o es muy pequeño. Si es el

caso, apelo a su conciencia. No les hagan a los demás lo que no les gustaría que le hicieran a ustedes. No produzcan una aplicación PHP conociendo todo lo que ahora conocen, solo porque luego no son ustedes los que van a dar el soporte. Eso es un acto de violencia casi física y gratuita.

- Si son profesores, decanos, asesores, CTOs o cualquier otra posición con poder de decisión sobre la tecnología a utilizar, piensen en la historia del lenguaje y sus herramientas. ¿Realmente están dispuestos a confiar sus recursos e invertir en una tecnología con estas características, incluso cuando los hayan corregido? ¿Es esta una plataforma confiable a largo plazo para proyectos complejos? ¿Es este el mejor uso que pueden hacer del tiempo de sus programadores? Las pérdidas en tiempo, frustración profesional y rendimiento de las aplicaciones están prácticamente garantizadas y son mayores que las necesarias para el entrenamiento de la fuerza de trabajo en otras tecnologías.
- Si se ven forzados por las circunstancias a considerar el uso de PHP, evítenlo en lo posible en favor de cualquier otra tecnología, y cuando esto no sea posible (que tampoco se pide fanatismo) usen frameworks, inviertan fuertemente en arquitectura y patrones de diseño y detallen minuciosamente las condiciones que los obligan a elegirlo, de manera que sea posible determinar cuando estas dejen de ser ciertas y poder reescribir el sistema lo más pronto posible. O al menos, saber porque cometieron tal error y salvar su reputación del juicio de los programadores futuros. Que siempre son muy críticos del trabajo que heredan…

Lean cosas, aprendan cosas, construyan cosas, diviértanse. **Y manténganse lejos de PHP**.

Referencias

- I'm sorry, but PHP sucks. – *https://maurus.net/resources/programming-languages/php/*
- Interview – PHP's Creator, Rasmus Lerdorf – *http://www.sitepoint.com/phps-creator-rasmus-lerdorf/*
- Small ISVs: You need Developers, not Programmers – *http://www.ericsink.com/No_Programmers.html*
- PHP: a fractal of bad design – *http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/*
- PHP Must Die – *http://www.steike.com/code/php-must-die/*
- Why PHP Sucks – *http://webonastick.com/php.html*
- PHP Sucks, But It Doesn't Matter – *http://www.codinghorror.com/blog/2008/05/php-sucks-but-it-doesnt-matter.html*
- The PHP singularity – *http://www.codinghorror.com/blog/2012/06/the-php-singularity.html*
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) – *http://www.joelonsoftware.com/articles/Unicode.html*
- What I don't like about PHP – *http://www.bitstorm.org/edwin/en/php/*
- In which I write about PHP for the first and the last time – *http://plasmasturm.org/log/393/*
- On PHP - *http://www.tbray.org/ongoing/When/200x/2006/02/17/PHP#p-21*
- PHP Sucks: So What? – *http://ilikekillnerds.com/2012/05/php-sucks-so-what/*
- Why PHP still sucks, even if you like it – *http://nerdlabor.de/blog/2012/04/13/why-php-still-sucks-even-if-you-like-it/*
- PHP Sucks! But I Like It! – *http://blog.ircmaxell.com/2012/04/php-sucks-but-i-like-it.html*
- Enough with the PHP Bashing, let's do a Reality Check – *http://blog.trumpetinteractive.com/post/35924589/enough-with-the-php-bashing-lets-do-a-reality-check*
- SIN NOMBRE – *http://news.ycombinator.com/item?id=4177516*