Ejercicios para el taller “Interfaces de scripting para librerias en C”

Autor: Moisés Humberto Silva Salmerón <moyhu@mx1.ibm.com>

Junto con este documento de ejercicios debe haber sido entregado un archivo ZIP con el código fuente completo de los mismos.

Los siguientes archivos deben existir en el archivo ZIP:

./ejercicios.pdf
./presentacion.ppt
./ejercicio3
./ejercicio3/ciisa.c
./ejercicio3/php_ciisa.h
./ejercicio3/config.m4
./ejercicio6
./ejercicio6/ext
./ejercicio6/ext/php_hpdf.h
./ejercicio6/ext/hpdf.c
./ejercicio6/ext/config.m4
./ejercicio6/creapdf_protegido.php
./ejercicio2
./ejercicio2/ciisa.c
./ejercicio2/php_ciisa.h
./ejercicio2/config.m4
./ejercicio5
./ejercicio5/creapdf_protegido.c
./ejercicio5/ext
./ejercicio5/ext/php_hpdf.h
./ejercicio5/ext/hpdf.c
./ejercicio5/ext/config.m4
./ejercicio5/creapdf_protegido.php
./ejercicio4
./ejercicio4/ciisa.c
./ejercicio4/php_ciisa.h
./ejercicio4/config.m4

Los ejercicios fueron probados usando PHP 5.2.4 y SWIG 1.1p5

Ejercicio 1. Compilación de PHP. (20 minutos)

El objetivo del ejercicio es familiarizarse con el arbol de código de PHP y su compilación desde la linea de comandos. Con esto lograremos un entendimiento de las partes que forman el intérprete de PHP y sus extensiones.

1. Descargar el código fuente de PHP en http://www.php.net/downloads.php#v5
2. Descomprimir el archivo mediante tar -xvpzf (si bajaste el .tar.gz) o tar -xvpjf (si bajaste el .tar.bz2)

```
# tar -xvpjf php-5.2.4.tar.bz2
```

Al descargar y descomprimir el tar, todo el código fuente del Zend Engine (ZE) y las extensiones de PHP está a tu disponibilidad. Dentro de la carpeta ext/ hay una sub-carpeta por cada extensión de PHP.

PHP utiliza un grupo de herramientas conocidas como “GNU Autotools” para su compilación. No todas las distribuciones de Linux incluyen por default estas herramientas. En el directorio raiz de PHP se encuentra un script generado por GNU autotools conocido como “configure”. El script de configure permite especificar opciones de compilación en la linea de comandos. Para ver las opciones soportadas podemos ejecutar el comando

```
# ./configure –help
```

Desde el directorio raiz de PHP. La salida debe mostrar todas las opciones que el script configure de PHP acepta para definir la forma en que el código fuente será compilado. La salida es bastante extensa y puede parecer intimidante, muchas opciones y poca ayuda para cada una. La buena noticias es que los defaults suelen ser suficiente en la mayoría de los casos. La opción que es de mayor interés para nosotros es –with-EXTENSION=[shared[,PATH]], la cual nos permite especificar extensiones que queremos sean compiladas. Por default el script configure intenterá determinar que extensiones pueden ser compiladas revisando si las librerias de las que las extensiones dependen están presentes en el sistema, sin embargo solo busca en los paths estandard del sistema como /usr/lib, si la libreria se encuentra en un path no estandar esta opción nos permite especificar el path hacia los headers y .so, también nos permite especificar si deseamos que la extensión sea compilada como módulo o internamente junto con el parser. La opción más recomendada y común es compilar las extensiones de PHP como módulos.

Podemos proceder a configurar nuestro ambiente para instalar PHP. Todo el proceso de instalación debe ser hecho por el usuario “root” de nuestro sistema.

```
# ./configure –prefix=/usr --enable-debug
```

Al terminar la ejecución del script de configure se habrá generado un Makefile para PHP, que es el encargado de invocar el compilador. Para ejecutar el Makefile y empezar la compilación escribimos el comando:

```
# make
```

Debido a que PHP puede funcionar como intérprete stand-alone o como interfaz hacia un servidor web via CGI, 2 intérpretes son compilados: cgi y cli. Durante este taller usaremos el intérprete de linea de comandos (CLI). Para comprobar que el intérprete fué compilado con éxito podemos

ejecutar el comando:

file sapi/cli/php

Debemos ver una salida como esta:

sapi/cli/php: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.9, dynamically linked (uses shared libs), for GNU/Linux 2.6.9, not stripped

Eso indica que el binario del intérprete fue compilado con éxito y puesto en el folder sapi/cli. Finalmente para completar la instalación ejecutamos el comando:

make install

Para comprobar la instalación ejecutamos:

php –version

PHP 5.2.4 (cli) (built: Jun 4 2008 21:00:32)
Copyright (c) 1997-2007 The PHP Group
Zend Engine v2.2.0, Copyright (c) 1998-2007 Zend Technologies

Ejercicio 2. CIISA Extension. (30 minutos)

El objetivo de este ejercicio es familiarizarnos con el ambiente de compilación y las declaraciones mínimas necesarias para crear una extensión de PHP antés de pasar a un ejemplo real.

1. Debemos iniciar por escoger un nombre para nuestra extensión. En este caso usaremos el nombre “ciisa”. Asi que creamos una subcarpeta en la carpeta ext/

```
# mkdir ext/ciisa
```

2. El primer archivo a crear es config.m4. Como vimos en el ejercicio 1 PHP utiliza el script “configure” para decidir como compilar los archivos. El archivo config.m4 es un script también que funcionará como extensión al script de configure de PHP para decidir cuando compilar nuestra extensión y como compilarla. El contenido de config.m4 debe ser:

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

if test "$PHP_CIISA" = "yes"; then
  AC_DEFINE(HAVE_CIISA, 1, [Whether you have CIISA])
  PHP_NEW_EXTENSION(ciisa, ciisa.c, $ext_shared)
fi
```

3. El macro PHP_NEW_EXTENSION define el nombre de la extensión y los archivos fuente que la componen, en este caso solamente ciisa.c, en caso de tener más archivos se separan por espacios solamente.

4. Después de crear config.m4, el primer archivo fuente que necesitamos es php_ciisa.h

```
#ifndef PHP_CIISA_H
#define PHP_CIISA_H 1

#define PHP_CIISA_VERSION "1.0"
#define PHP_CIISA_EXTNAME "ciisa"

PHP_FUNCTION(print_ciisa_date);

extern zend_module_entry ciisa_module_entry;
#define phpext_ciisa_ptr &ciisa_module_entry

#endif
```

La linea PHP_FUNCTION es el prototipo de la función que vamos a crear. La estructura ciisa_module_entry será usada por el Zend Engine para obtener información sobre las funciones exportadas al usuario y de inicialización y destrucción del módulo.

4. Finalmente el cuerpo del archivo en ciisa.c (debe coincidir el nombre con el los fuentes registrados en config.m4). El cuerpo del archivo consiste en los include:

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

#include "php.h"
#include "php_ciisa.h"
```

php.h contiene APIs que necesitaremos durante la implementación de la extensión para comunicarnos con el ZE y php_ciisa.h es el header que nosotros creamos para nuestras propias declaraciones.

5. Continuamos con las declaraciones necesarias para que el ZE cargue la extensión.

```
static function_entry ciisa_functions[] = {
	PHP_FE(print_ciisa_date, NULL)
	{NULL, NULL, NULL}
};

zend_module_entry ciisa_module_entry = {
	STANDARD_MODULE_HEADER,
	PHP_CIISA_EXTNAME,
	ciisa_functions,
	NULL,
	NULL,
	NULL,
	NULL,
	NULL,
	PHP_CIISA_VERSION,
	STANDARD_MODULE_PROPERTIES
};
```

ciisa_functions es un arreglo de estructuras donde cada elemento define una función, su manejador y argumentos. En el caso de funciones el segundo argumento a PHP_FE se deja nulo ya que los argumentos se parsean al momento de ejecución por la misma función no por el ZE.

ciisa_module_entry es una estructura con varios apuntadores a funciones y un lugar para registrar el arreglo ciisa_functions que determina todas las funciones de nuestra extensión.

6. Finalmente tenemos la implementación de la función.

```
PHP_FUNCTION(print_ciisa_date)
{
   php_printf("Septiembre del 2008\n");
}
```

7. Para finalmente compilar la extensión usamos el comando “phpize” que es instalado junto con el intérprete.

```
                # phpize
Configuring for:
PHP Api Version:         20041225
Zend Module Api No:      20060613
Zend Extension Api No:   220060519
```

8. El comando debe haber generado varios archivos para el ambiente de compilación. Ahora podemos ejecutar el script de configure de nuestra extensión y el comando “make” para compilar.

```
# ./configure –with-ciisa
# make
```

9. Al terminar la compilación debemos tener el módulo ciisa.so en la carpeta modules/ del directorio donde tenemos los archivos de la extensión. Finalmente comprobamos que los hooks que necesita el ZE esten presentes.

```
# objdump -T ciisa.so | grep module
modules/ciisa.so:     file format elf32-i386
00000470 g    DF .text  00000016  Base        get_module
00001660 g    DO .data  00000058  Base        ciisa_module_entry
```

El símbolo get_module fué generado debido al macro ZEND_GET_MODULE(ciisa) en ciisa.c, ciisa_module_entry es la estructura que define la extensión, y es el valor que regresa get_module() al Zend Engine.

10. Ahora podemos probar nuestra extensión desde un script PHP. Primero determinamos la ubicación del archivo php.ini que utiliza nuestro intérprete.

```
# php -i | grep -i loaded
Loaded Configuration File => /usr/lib/php.ini
```

11. Editamos php.ini para especificar el directorio de las extensiones y solicitar la carga inmediata de nuestra extensión.

```
extension_dir=/usr/lib/php/extensions/
extension=ciisa.so
```

12. Creamos un script PHP que utilice nuestra función.

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

13. Se ejecuta para verificar el resultado.

```
# php -f script.php
Septiembre del 2008
```

Ejercicio 3. Ciclos de vida. (20 minutos)

Este pequeño ejercicio nos ayudará a comprender los ciclos de vida de una extensión PHP. Algunos miembros de la estructura de registro ciisa_module_entry quedaron nulos en el ejercicio anterior. Veamos el significado y uso de cada uno.

```
zend_module_entry ciisa_module_entry = {
    STANDARD_MODULE_HEADER,
    PHP_CIISA_EXTNAME,
    ciisa_functions,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    PHP_CIISA_VERSION,
    STANDARD_MODULE_PROPERTIES
};
```

El primer miembro es STANDARD_MODULE_HEADER. En realidad este macro define los primeros 6 miembros de zend_module_entry, y para nuestros propósitos este es el valor adecuado siempre para crear una extensión.

El séptimo miembro PHP_CIISA_EXTNAME es un char* al nombre de la extensión.

El octavo miembro es un arreglo de estructuras que definen las funciones exportadas por la extensión.

El noveno miembro, que fue especificado nulo es una función que será llamada al cargarse el módulo y se le conoce como el miembro MINIT. El décimo miembro es la contraparte, MSHUTDOWN, una apuntador a una función que será llamada al descargar el módulo.

Los miembros once y doce son también apuntadores a funciones, conocidos como RINIT y RSHUTDOWN. Para el caso del intérprete de linea de comandos esta función no tiene diferencia contra MINIT y MSHUTDOWN. Sin embargo, en un ambiente web, donde la misma instancia del intérprete puede ser usada por multiples requests, RINIT es llamada al inicio de un request, RSHUTDOWN al termino del request. Para mostrar su uso seguimos los siguientes pasos:

1. Editamos php_ciisa.h para agregar los 4 prototipos (MINIT, MSHUTDOWN, RINIT, RSHUTDOWN)

```
PHP_MINIT_FUNCTION(ciisa);
PHP_MSHUTDOWN_FUNCTION(ciisa);
PHP_RINIT_FUNCTION(ciisa);
PHP_RSHUTDOWN_FUNCTION(ciisa);
```

2. Implementamos las funciones en ciisa.c

```
PHP_MINIT_FUNCTION(ciisa)
{
	php_printf("Cargando modulo CIISA ...\n");
}

PHP_MSHUTDOWN_FUNCTION(ciisa)
{
	php_printf("Descargando modulo CIISA ...\n");
}

PHP_RINIT_FUNCTION(ciisa)
{
	php_printf("Atendiendo request!\n");
}

PHP_RSHUTDOWN_FUNCTION(ciisa)
{
	php_printf("Finalizando request!\n");
}
```

3. Agregamos apuntadores a las funciones en ciisa_module_entry

```
zend_module_entry ciisa_module_entry = {
	STANDARD_MODULE_HEADER,
	PHP_CIISA_EXTNAME,
	ciisa_functions,
	PHP_MINIT(ciisa),
	PHP_MSHUTDOWN(ciisa),
	PHP_RINIT(ciisa),
	PHP_RSHUTDOWN(ciisa),
	NULL,
	PHP_CIISA_VERSION,
	STANDARD_MODULE_PROPERTIES
};
```

4. Compilamos nuevamente la extensión y corremos el script.

```
	# make
	# php -f script.php
	Cargando modulo CIISA ...
	Atendiendo request!
	Septiembre del 2008
	Finalizando request!
	Descargando modulo CIISA ...
```

El órden de ejecución es MINIT, RINIT, RSHUTDOWN, MSHUTDOWN. RINIT y RSHUTDOWN se repetirán cuando el intérprete está en un servidor web e incluso puede haber varios hilos de ejecución simultaneos para RINIT y RSHUTDOWN.

Ejercicio 4. Recepción de argumentos y retorno de valores. (30 minutos)

Durante este ejercicio aprenderemos a retornar distintos tipos de valores a los usuarios de nuestra extensión y usaremos una API del ZE para recibir argumentos.

1. Creamos una nueva función get_ciisa_date

```
PHP_FUNCTION(get_ciisa_date)
{
    RETURN_STRING("Septiembre del 2008\n", 1);
}
```

2. El macro RETURN_STRING modifica el zval (estructura que representa una variable internamente en PHP) de retorno para pasarlo al usuario. El número 1 del segundo argumento indica si el Zend Engine debe duplicar el string. Cualquier string que no haya sido solicitado con las funciones de memoria del ZE como emalloc() debe ser duplicado para evitar una violación de segmento. De la misma forma que existe el macro RETURN_STRING, existen macros para regresar otro tipo de valores al usuario:

```
RETURN_BOOL(b)
RETURN_NULL()
RETURN_LONG(l)
RETURN_DOUBLE(d)
RETURN_ZVAL ...
```

3. Creamos una nueva función que reciba un string como argumento.

```
PHP_FUNCTION(ciisa_says)
{
    char *message;
    int len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
        "s", &message, &len) == FAILURE) {
        RETURN_FALSE;
    }
    php_printf("CIISA says: %s\n", message);
    RETURN_TRUE;
}
```

La API zend_parse_parameters tomará los argumentos del stack del Zend VM (Virtual Machine) y asignará sus direcciones a las variables que recibe. En este caso, la cadena de formato es una simple “s” que indica que el primer y único argumento es una cadena. Por cada “s” en la cadena de formato se debe proveer de un char** y un int* para que el ZE los llene con la dirección de la cadena y su longitud.

La cadena de formato acepta los siguientes caracteres:

Tipo User Space	Caracter	Tipo de variable interno
Bool	b	zend_bool
Long	Ll	long
Double	Dd	double
String	s	char*, int
Resource	r	zval*
Array	a	zval*
Object	o	zval*
Opaco	z	zval*

zval es una estructura que representa una variable dentro del ZE. He aqui unas definiciones importantes tomadas de Zend/zend.h

```
typedef struct _zval_struct zval
typedef union _zvalue_value {
        long lval;                              /* long value */
        double dval;                    /* double value */
        struct {
                char *val;
                int len;
        } str;
        HashTable *ht;                  /* hash table value */
        zend_object_value obj;
} zvalue_value;

struct _zval_struct {
        /* Variable information */
        zvalue_value value;             /* value */
        zend_uint refcount;
        zend_uchar type;        /* active type */
        zend_uchar is_ref;
};
```

La estructura zval tiene la informacion de:

- Valor de la variable a través del miembro “value” de tipo zvalue_value.
- Conteo de referencias a la variable mediante zend_uint refcount.
- Tipo actual de la variable mediante zend_uchar type.
- Indicador de si la variable es una referencia o la variable en si misma mediante zend_uchar is_ref.

Dado que las variables en PHP pueden tomar un valor distinto a cada momento, el miembro type es muy importante para determinar cual valor debe considerarse. Los valores que puede contener el miembro type son:

Macro	Valor
IS_NULL	0
IS_LONG	1
IS_DOUBLE	2
IS_BOOL	3
IS_ARRAY	4
IS_OBJECT	5
IS_STRING	6
IS_RESOURCE	7
IS_CONSTANT	8
IS_CONTSTANT_ARRAY	9
IS_CONSTANT_INDEX	0x80

La unión zvalue_value a su vez contiene el valor en si mismo de la variable.

Ejercicio 5. Crear una interfaz en PHP para la librería HPDF. (40 minutos)

La libreria HPDF es una librería en C para crear documentos PDF, desde lo mas simple a lo mas complejo. Actualmente existen distintos modos de crear documentos PDF desde PHP, sin embargo el uso de la librería HPDF desde PHP nos permitirá mostrar claramente como puede crearse una interfaz de scripting para una librería en C.

Dado que la librería HPDF es muy extensa, para fines ilustrativos solo exportaremos las funciones necesarias para duplicar un programa ya existente en C. El programa de prueba que deseamos escribir ahora en PHP se encuentra en la carpeta de ejercicio5 y su nombre es creapdf_protegido.c

Podemos compilar creapdf_protegido con:

```
#  gcc -Wall -lhpdf -DHPDF_SHARED creapdf_protegido.c -o creapdf_protegido.exe
```

Ahora para mostrar su uso lo ejecutamos asi:

```
# ./creapdf_protegido.exe nuevo "PDF creado al vuelo en el CIISA" miclave otraclave
```

El programa crea un PDF con el nombre nuevo.pdf que tendrá como contenido el texto “PDF creado al vuelo en el CIISA” usando como passwords de propietario y usuario “miclave” y “otraclave” respectivamente.

Para crear el mismo programa desde PHP necesitamos crear una nueva extensión que use la librería. Empezamos por crear un nuevo directorio en el arbol de extensiones de PHP.

```
# mkdir ext/hpdf/
```

Al igual que con nuestra primera extensión, tenemos que crear 3 archivos. El archivo de configuración para la compilación (config.m4), el archivo C principal hpdf.c y su header php_hpdf.h.

Existe una diferencia importante entre esta nueva extensión y la extensión de ciisa. En esta extensión tenemos una dependencia en una libería externa, que suele ser el caso de muchas extensiones. Debemos reflejar esta dependencia en config.m4 para que la compilación encuentre el header de hpdf.h y la librería para poder enlazarla. De igual forma, dentro de config.m4 debemos tratar de encontrar el directorio donde se ubican los headers y binarios de la libería.

```
for i in $PHP_HPDF /usr/local /usr
do
        if test -r $i/include/hpdf.h
        then
                HPDF_DIR="$i"
                HPDF_INC_DIR="$i/include"
                break
        fi
done

if test -z "$HPDF_DIR"
then
        AC_MSG_ERROR([No pude encontrar los headers de HPDF en $PHP_HPDF])
fi
```

```
for i in $HPDF_DIR $PHP_LIBDIR
do
	if test -r $i/lib/libhpdf.so
	then
		HPDF_LIB_DIR="$i/lib"
		break
	fi
done

if test -z "$HPDF_LIB_DIR"
then
	AC_MSG_ERROR(no pude encontrar libhpdf.so en $HPDF_DIR)
fi

PHP_ADD_INCLUDE($HPDF_INC_DIR)
PHP_ADD_LIBRARY_WITH_PATH($HPDF_LIBNAME, $HPDF_LIB_DIR, HPDF_SHARED_LIBADD)
PHP_SUBST(HPDF_SHARED_LIBADD)
PHP_NEW_EXTENSION(hpdf, hpdf.c, $ext_shared,,"-DHPDF_SHARED")
```

El script config.m4 acepta la sintaxis del shell de Linux, por lo que podemos usar un bucle para buscar el header requerido y la librería antés de la compilación. En base a lo que se encuentre se usan los macros de PHP para instruir al script de compilación las dependencias.

Una vez hecho el config.m4 podemos proceder a seguir el mismo patrón que seguimos para la extensión ciisa. La diferencia es que ahora la implementación de las funciones serán simplemente wrappers para las funciones de C de la librería hpdf.

El primer problema con que nos topamos es que la API HPDF_New regresa un handle. Como representamos un handle dentro de PHP de tal forma que podamos asociarlo de nuevo en llamadas a API's subsecuentes? La respuesta es el tipo de dato “resource” de PHP. Los resources son tipos de dato genéricos para representar un recurso x. Por ejemplo, una conexión a una base de datos. En este caso necesitamos 3 resources, uno para HPDF_Doc, otro para HPDF_Page y uno último para HPDF_Font.

Para registrar los resources requerimos de 3 cosas.

- Una variable global de nuestra extensión que representará una lista de recursos de cierto tipo.
- Registrar cada tipo de recurso mediante zend_register_list_destructors_ex.
- Registrar cada nuevo recurso mediante ZEND_REGISTER_RESOURCE.
- Crear una estructura interna por cada tipo de resource.

La variable global se puede registrar en cualquier punto de hpdf.c, es costumbre de las extensiones de PHP iniciar el nombre de estas variables con el prefijo “le” (list entry).

```
int le_hpdf_doc;
int le_hpdf_page;
int le_hpdf_font;
```

La llamada a zend_register_list_destructors_ex debe hacerse naturalmente dentro de MINIT, dado que los recursos deben estar disponibles para el usuario.

```
static void php_hpdf_doc_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
	php_hpdf_doc_t *doc = (php_hpdf_doc_t *)rsrc->ptr;
	if (doc) {
		HPDF_Free(doc->handle);
		efree(doc);
	}
}

PHP_MINIT_FUNCTION(hpdf)
{
	le_hpdf_doc = zend_register_list_destructors_ex(php_hpdf_doc_dtor, NULL,
PHP_HPDF_DOC_RES_NAME, module_number);
	le_hpdf_page = zend_register_list_destructors_ex(NULL, NULL,
PHP_HPDF_PAGE_RES_NAME, module_number);
	le_hpdf_font = zend_register_list_destructors_ex(NULL, NULL,
PHP_HPDF_FONT_RES_NAME, module_number);
	REGISTER_LONG_CONSTANT("HPDF_PAGE_SIZE_B5", HPDF_PAGE_SIZE_B5,
CONST_PERSISTENT | CONST_CS);
	REGISTER_LONG_CONSTANT("HPDF_PAGE_LANDSCAPE",
HPDF_PAGE_LANDSCAPE, CONST_PERSISTENT | CONST_CS);
}
```

zend_register_list_destructors_ex espera dos apuntadores a funciones que determinan los destructores (persistente y no-persistente) de los recursos del tipo en cuestión. En nuestro caso unicamente registramos destructor no-persistente para HPDF_Doc, debido a que la interfaz de la librería solo requiere liberar ese recurso en particular.

La llamada a REGISTER_LONG_CONSTANT, registra una constante y su valor para que pueda ser usada como constante desde PHP.

El siguiente paso es, cada vez que un nuevo recurso sea creado, debemos asociarlo a la lista de su tipo. Por eso las llamadas a HPDF_New, HPDF_GetFont y HPDF_AddPage generan un recurso, lo registran y regresan al usuario mediante ZEND_REGISTER_RESOURCE.

```
PHP_FUNCTION(HPDF_New)
{
	php_hpdf_doc_t *doc = emalloc(sizeof(php_hpdf_doc_t));
	doc->handle = HPDF_New(error_handler, NULL);
	if (!doc->handle) {
		efree(doc);
		php_printf("error: HPDF_New\n");
		RETURN_FALSE;
	}
	ZEND_REGISTER_RESOURCE(return_value, doc, le_hpdf_doc);
}
```

La mayoría de las extensiones en PHP que manejan recursos lo hacen de este modo. Primero llaman emalloc() para solicitar memoria manejada para el recurso, le asignan un valir y finalmente mandan llamar ZEND_REGISTER_RESOURCE. Este macro agregará el recurso a la lista apropiada y se lo asignará al zval que se le indique.

Finalmente, aunque el ZE mandará llamar el destructor de recursos registrados al finalizar el script, podemos proveer al usuario de una forma de liberar el recurso explicitamente mediante HPDF_Free.

```
PHP_FUNCTION(HPDF_Free)
{
	zval* zdoc;
	php_hpdf_doc_t *doc;
	if (FAILURE == zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r", &zdoc)){
		RETURN_FALSE;
	}
	ZEND_FETCH_RESOURCE(doc, php_hpdf_doc_t*, &zdoc, -1,
PHP_HPDF_DOC_RES_NAME, le_hpdf_doc);
	/* ZEND_FETCH_RESOURCE llama RETURN_FALSE si el resource no es del tipo requerido
	  por lo que en este punto ya sabemos que tenemos un resource tipo HPDF_Doc valido */
	zend_list_delete(Z_LVAL_P(zdoc));
	RETURN_TRUE;
}
```

Podemos notar al menos un par de cosas nuevas en este trozo de código. Primero, para recibir como argumento a una función un recurso, lo hacemos con la letra “r” en la cadena de formato para zend_parse_parameters y el valor asignado es un zval*. Inmediatamente después usamos el macro ZEND_FETCH_RESOURCE para solicitar a PHP la validación del recurso y estar seguros de que el tipo de recurso es el que deseamos. ZEND_FETCH_RESOURCE terminará la ejecucion de la función y regresará falso si el recurso no es adecuado. Por último, llamamos la función zend_list_delete que se encarga de eliminar el recurso de la lista. El macro Z_LVAL_P es simplemente para tomar el valor IS_LONG de un zval*, los recursos son guardados en el zval* simplemente con un identificador de tipo long, es por ello que comparten el uso del macro Z_LVAL_P con los valores explicitamente de tipo IS_LONG.

Finalmente, podemos usar el script creapdf_protegido.php, para corroborar que nuestra extensión funciona adecuadamente y genera un PDF protegido por password de la misma forma que el programa en C.

Ejercicio 6. Crear una interfaz OO en PHP para la librería HPDF. (40 minutos)

En este último ejercicio crearemos una interfaz orientada a objetos en PHP para la API en C de la libreria HPDF.

PHP 5 introduce importantes mejoras en la orientación a objetos. Para tomar ventaja de ellas es común cambiar la interfaz de código estructurado C por una nueva interfaz moderna orientada a objetos desde un lenguaje de scripting como PHP. Para crear nuestra interfaz orientada a objetos re-utilizaremos el archivo config.m4, ya que la configuración de la compilacion sigue siendo la misma. Podemos usar como base hpdf.c y php_hpdf.h, sin embargo se requieren cambios en algunas declaraciones.

El primer cambio importante es que nos deshacemos del arreglo estático function_entry donde teniamos declaradas todas las funciones de nuestra extensión. La razón es que ahora expondremos clases con métodos y no funciones. De igual forma tenemos que quitar el apuntador a este arreglo de la estructura zend_module_entry.

```
zend_module_entry hpdf_module_entry = {
    STANDARD_MODULE_HEADER,
    PHP_HPDF_EXTNAME,
    NULL,
    PHP_MINIT(hpdf),
    NULL,
    NULL,
    NULL,
    NULL,
    PHP_HPDF_VERSION,
    STANDARD_MODULE_PROPERTIES
};
```

Así ahora tenemos NULL en en el lugar del apuntador al arreglo de funciones. Entonces como hacemos ahora para exponer las clases? El primer paso es decidir cuantas clases vamos a exponer. En nuestro caso expondremos 2 clases.

La clase HPDF expondrá la funcionalidad del tipo HPDF_Doc y la clase HPDFPage expondrá la funcionalidad del tipo HPDF_Page. El recurso de tipo font continuará existiendo tal como antés.

Ahora que hemos decidido como exponer la funcionalidad, podemos proceder a declarar dos estructuras de tipo zend_function_entry, que listará los métodos de cada clase.

Primero iniciamos con los de la clase HPDF.

```
zend_function_entry HPDF_methods[] = {
	PHP_ME(HPDF, __construct, NULL, ZEND_ACC_PUBLIC | ZEND_ACC_CTOR)
	PHP_ME(HPDF, GetFont, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDF, AddPage, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDF, SetPassword, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDF, SaveToFile, NULL, ZEND_ACC_PUBLIC)
	{ NULL, NULL, NULL }
};
```

Es muy similar a la definición de las funciones, con la diferencia de que especificamos primero el nombre de la clase a la que pertenecerá la función (dado que será un método para los usuarios) y podemos especificar algunas características propias de los métodos como su visibilidad (ZEND_ACC_PUBLIC).

Ahora los métodos de HPDFPage.

```
zend_function_entry HPDFPage_methods[] = {
	PHP_ME(HPDFPage, __construct, NULL, ZEND_ACC_PUBLIC | ZEND_ACC_CTOR)
	PHP_ME(HPDFPage, SetSize, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, BeginText, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, SetFontAndSize, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, MoveTextPos, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, ShowText, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, EndText, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, GetWidth, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, GetHeight, NULL, ZEND_ACC_PUBLIC)
	PHP_ME(HPDFPage, TextWidth, NULL, ZEND_ACC_PUBLIC)
	{ NULL, NULL, NULL }
};
```

Igual que con el listado de funciones se requiere un trio de NULL's al final. Ahora que tenemos la declaración de los métodos necesitamos crear el cuerpo de cada método. Podemos re-utilizar las funciones que ya teniamos del ejercicio 5 con algunos ajustes.

Veamos como quedaría el constructor de HPDF (reemplaza a la función HPDF_New).

```
PHP_METHOD(HPDF, __construct)
{
	HPDF_Doc doc = HPDF_New(error_handler, NULL);
	if (!doc) {
		zend_throw_exception(zend_get_error_exception(), "HPDF_New failed", -1 TSRMLS_CC);
		return;
	}
	zend_update_property_long(EG(scope), getThis(), "__handle", sizeof("__handle"), (long)doc TSRMLS_CC);
}
```

Podemos notar que la primer diferencia es el uso del macro PHP_METHOD en lugar de PHP_FUNCTION. El macro recibe el nombre de la clase a la que pertenece la función-método y el nombre del mismo. El método es muy simple, unicamente usa la función HPDF_New para crear un nuevo handle a un documento PDF y guarda el handle en una propiedad privada de la clase llamada __handle. Los usuarios naturalmente no pueden modificar o leer el valor de las propiedades privadas por lo que no es un problema. Mas adelante veremos como creamos propiedades de las clases.

El segundo cambio notorio es el uso de zend_throw_exception. Esta función arroja una excepción al ambiente del usuario. Por último getThis() es un macro útil para obtener el zval* del objeto del usuario que está ejecutando el método.

Todos los métodos serán similares, solo hay que recuperar la propiedad en donde guardamos el handle y llamar las funciones de la librería directamente para regresar un resultado en caso de ser necesario. El único método que difiere un poco es HPDF::AddPage, debido a que debe regresar un objeto de tipo HPDFPage, veamos como funciona.

```
PHP_METHOD(HPDF, AddPage)
{
	HPDF_Page page;
	HPDF_Doc doc = get_hpdf_handle(getThis());
	page = HPDF_AddPage(doc);
	if (!page) {
		zend_throw_exception(zend_get_error_exception(), "HPDF_AddPage error", -1 TSRMLS_CC);
		return;
	}
	object_init_ex(return_value, HPDFPage_ce);
	return_value->refcount = 1;
	return_value->is_ref = 1;
	zend_update_property_long(HPDFPage_ce, return_value, "__handle", sizeof("__handle"), (long)page TSRMLS_CC);
}
```

El método llama la API HPDF_AddPage para crear un handle de tipo HPDF_Page. No podemos regresar este handle porque deseamos que se use la interfaz orientada a objetos. Así que necesitamos construir un objeto de tipo HPDFPage y asignarle el handle a una propiedad privada del mismo. Aqui entra la variable return_value. La variable return_value está siempre presente en todos los métodos y funciones de una extensión y es de tipo zval*, es decir, una variable del usuario. Todos los métodos en PHP devuelven un valor, por default el miembro “type” de zval es 0, es decir, IS_NULL. A menos que el cuerpo del método o la función cambie este hecho el valor de retorno será NULL, todas las funciones o métodos de PHP por default devuelven NULL. Nosotros deseamos regresar un objeto asi que llamamos la función object_init_ex() pasandole el zval* return_value y HPDFPage_ce para que se cambie el tipo del zval* de IS_NULL a IS_OBJECT y se inicialicen todos los miembros de HPDFPage. Inicializamos manualmente su refcount a 1 y su is_ref a 1, dado que en PHP5 todos los

objetos son en realidad referencias. Finalmente se actualiza su propiedad privada __handle con el valor del handle que nos devolvió HPDF_AddPage.

Ya que tenemos todos nuestros métodos, solo resta registrar las clases con su lista de métodos y cambiar nuestras constantes globales del ejercicio anterior a constantes de clase. Ambas cosas se hacen en la función MINIT (carga de módulo).

```
PHP_MINIT_FUNCTION(hpdf)
{
	zend_class_entry hpdf_class;
	zend_class_entry hpdfpage_class;

	INIT_CLASS_ENTRY(hpdf_class, "HPDF", HPDF_methods);
	HPDF_ce = zend_register_internal_class(&hpdf_class TSRMLS_CC);
	zend_declare_property_long(HPDF_ce, "__handle", sizeof("__handle"), -1, ZEND_ACC_PRIVATE TSRMLS_CC);

	INIT_CLASS_ENTRY(hpdfpage_class, "HPDFPage", HPDFPage_methods);
	HPDFPage_ce = zend_register_internal_class(&hpdfpage_class TSRMLS_CC);
	zend_declare_property_long(HPDFPage_ce, "__handle", sizeof("__handle"), -1, ZEND_ACC_PRIVATE
TSRMLS_CC);
	zend_declare_class_constant_long(HPDFPage_ce, "SIZE_B5", sizeof("SIZE_B5")-1, HPDF_PAGE_SIZE_B5
TSRMLS_CC);
	zend_declare_class_constant_long(HPDFPage_ce, "LANDSCAPE", sizeof("LANDSCAPE")-1,
HPDF_PAGE_LANDSCAPE TSRMLS_CC);

	le_hpdf_font = zend_register_list_destructors_ex(NULL, NULL, PHP_HPDF_FONT_RES_NAME, module_number);
}
```

El registro del recurso para HPDF_Font sigue siendo exactamente igual que el ejercicio anterior. Para registrar las clases primero inicializamos una estructura de tipo zend_class_entry usando INIT_CLASS_ENTRY especificando el nombre de nuestra clase (HPDF y HPDFPage) y el arreglo en donde sus métodos están definidos. Posteriormente hacemos el registro mediante zend_register_internal_class y el Zend Engine nos regresará un apuntador al identificador de la clase (en realidad una estructura del mismo tipo zend_class_entry).

Durante la definición de los métodos estuvimos llevando registro interno de los handles a HPDF_Doc y HPDF_Page mediante propiedades privadas de la clase. Este es el momento de registrar esas propiedades usando zend_declare_property_long() (Ya que los handles de la libreria son void*, un long funciona perfectamente para guardar el valor).

Finalmente para crear constantes de clase usamos zend_declare_class_constant para que los usuarios puedan usar HPDFPage::SIZE_B5 y HPDFPage::LANDSCAPE desde sus scripts de PHP. Para corroborar, en los archivos del taller (carpeta ejercicio6) se incluye el script creapdf_protegido.php en formato orientado a objetos.