# **Лабораторная №9. Конструкции языка PHP (часть 2). Функции. Ссылки**

## **Конструкции возврата значений**

### **Конструкция return**

Конструкция **return** возвращает значения, преимущественно из пользовательских функций, как параметры функционального запроса. При вызове **return** исполнение пользовательской функции прерывается, а конструкция **return** возвращает определенные значения.

Если конструкция **return** будет вызвана из глобальной области определения (вне пользовательских функций), то скрипт также завершит свою работу, а **return** также возвратит определенные значения.

Преимущественно, конструкция **return** используется для возврата значений пользовательскими функциями. Возвращаемые значения могут быть любого типа, в том числе это могут быть списки и объекты. Возврат приводит к завершению выполнения функции и передаче управления обратно к той строке кода, в которой данная функция была вызвана.

Пример использования конструкции **return** для возврата значений типа **integer**:

<?php  
function retfunct()  
{  
     return 7;  
}  
echo retfunct();   // выводит '7'.  
?>

Пример возврата конструкцией **return** массивов:

<?php

function numbers()

{

return array (0, 1, 2);

}

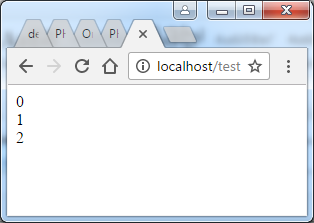
list ($zero, $one, $two) = numbers();

echo $zero, "<br>";

echo $one, "<br>";

echo $two;

?>



Для того, чтобы функция возвращала результат **по ссылке**, вам необходимо использовать оператор **&** и при описании функции, и при присвоении переменной возвращаемого значения:

<?php  
function &returns\_reference()  
{  
   return $someref;  
}  
  
$newref =& returns\_reference();  
?>

Как мы видим, конструкция **return** весьма удобна для применения в пользовательских функциях.

## **Конструкции включений в PHP**

Конструкции включений позволяют собирать PHP программу (скрипт) из нескольких отдельных файлов.

В PHP существуют две основные конструкции включений: **require** и **include**.

### **Конструкция включений require**

Конструкция **require** позволяет включать файлы в сценарий PHP до исполнения сценария PHP. Общий синтаксис **require** такой:

require имя\_файла;

При запуске (именно при запуске, а не при исполнении!) программы интерпретатор просто заменит инструкцию на содержимое файла **имя\_файла** (этот файл может также содержать сценарий на PHP, обрамленный, как обычно, тэгами **<?** и**?>**). Причем сделает он это непосредственно перед запуском программы (в отличие от **include**, который рассматривается ниже). Это бывает довольно удобно для включения в вывод сценария различных шаблонных страниц HTML-кодом. Приведем пример:

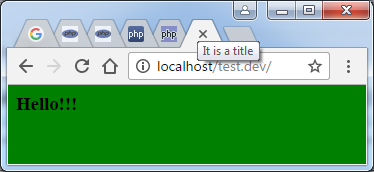
**Файл header.html:**  
<html>  
<head><title>It is a title</title></head>  
<body bgcolor=green>

**Файл footer.html:**  
h3>Hello!!!</h3>

</body></html>

**Файл script.php**  
<?php  
require "header.html";  
*// Сценарий выводит само тело документа*  
require "footer.html";  
?>

Таким образом, конструкция **require** позволяет собирать сценарии PHP из нескольких отдельных файлов, которые могут быть как html-страницами, так и php-скриптами.



Конструкция **require** поддерживает включения удаленных файлов (начиная с версии PHP 4.3.0). Например:

<?php   
*// Следующий пример не работает, поскольку пытается включить локальный файл*  
require 'file.php?foo=1&bar=2';  
*// Следующий пример работает*  
require 'http://www.example.com/file.php?foo=1&bar=2';  
?>

Конструкция **require** позволяет включать удаленные файлы, если такая возможность включена в конфигурационном файле PHP.

### **Конструкция включений include**

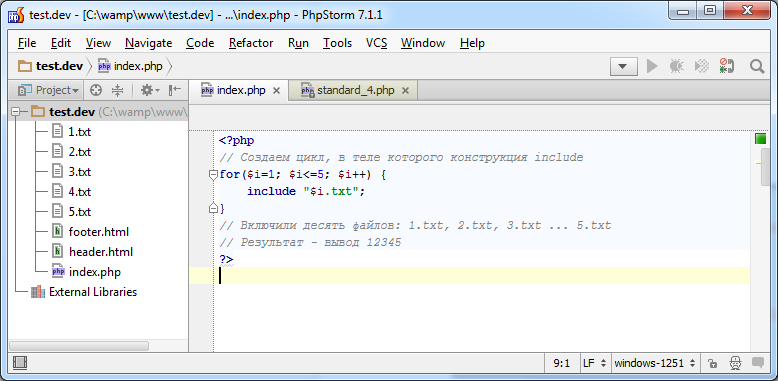
Конструкция **include** также предназначена для включения файлов в код сценария PHP.

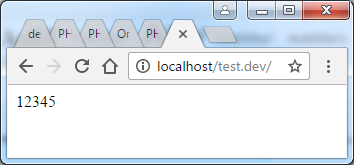
В отличие от конструкции **require** конструкция **include** позволяет включать файлы в код PHP скрипта во время выполнения сценария. Синтаксис конструкции **include** выглядит следующим образом:

include имя\_файла;

Поясним принципиальную разницу между конструкциями **require** и **include** на конкретном практическом примере. Создадим 10 файлов с именами 1.txt, 2.txt и так далее до 5.txt, содержимое этих файлов – просто десятичные цифры 1, 2 ...… 5 (по одной цифре в каждом файле). Создадим такой сценарий PHP:

<?php  
*// Создаем цикл, в теле которого конструкция include*  
for($i=1; $i<=5; $i++) {  
include "$i.txt";  
}  
*// Включили десять файлов: 1.txt, 2.txt, 3.txt ... 5.txt  
// Результат - вывод 12345*  
?>





В результате мы получим вывод, состоящий из 10 цифр: "12345678910". Из этого мы можем сделать вывод, что каждый из файлов был включен по одному разу прямо во время выполнения цикла! Если мы поставим теперь вместо **include** **require**, то сценарий сгенерирует критическую ошибку (fatal error).

PHP преобразует сценарий во внутреннее представление, анализируя строки сценария по очереди, пока не доходит до конструкции **include**. Дойдя до **include**, PHP прекращает транслировать сценарий и переключается на указанный в **include** файл. Таким образом из-за подобного поведения транслятора, быстродействие сценария снижается, особенно при большом количестве включаемых с помощью **include** файлов. С **require** таких проблем нет, поскольку файлы с помощью **require** включаются до выполнения сценария, то есть на момент трансляции файл уже включен в сценарий.

Таким образом, целесообразнее использовать конструкцию **require** там, где не требуется динамическое включение файлов в сценарий, а конструкцию **include** использовать только с целью динамического включения файлов в код PHP скрипта.

Конструкция **include** поддерживает включения удаленных файлов (начиная с версии PHP 4.3.0). Например:

<?php  
*// Следующий пример не работает, поскольку пытается включить локальный файл*  
include 'file.php?foo=1&bar=2';  
*// Следующий пример работает*  
include 'http://www.example.com/file.php?foo=1&bar=2';  
?>

Конструкция **include** позволяет включать удаленные файлы, если такая возможность включена в конфигурационном файле PHP.

### **Конструкции однократного включения require\_once и include\_once**

В больших PHP сценариях инструкции **include** и **require** применяются очень часто. Поэтому становится довольно сложно контролировать, как бы случайно не включить один и тот же файл несколько раз, что чаще всего приводит к ошибке, которую сложно обнаружить.

В PHP предусмотрено решение данной проблемы. Используя конструкции однократного включения **require\_once** и **include\_once**, можно быть уверенным, что один файл не будет включен дважды. Работают конструкции однократного включения **require\_once** и **include\_once** так же, как и **require** и **include** соответственно. Разница в их работе лишь в том, что перед включением файла интерпретатор проверяет, включен ли указанный файл ранее или нет. Если да, то файл не будет включен вновь.

Конструкции однократных включений также **require\_once и include\_ince** также позволяют включать удаленные файлы, если такая возможность включена в конфигурационном файле PHP.

## **Ссылки в PHP**

Хотя в PHP нет такого понятия, как указатель, все же существует возможность создавать ссылки на другие переменные. Существует две разновидности сылок: **жесткие** и **символические** часто называют просто ссылками). Жесткие ссылки появились в PHP версии 4 (в третьей версии существовали лишь символические ссылки).

**Ссылки в PHP** – это средство доступа к содержимому одной переменной под разными именами. Они не похожи на указатели языка Си и не являются псевдонимами таблицы символов. В PHP имя переменной и её содержимое – это разные вещи, поэтому одно содержимое может иметь разные имена.

### **Жесткие ссылки в PHP**

**Жесткая ссылка** представляет собой просто переменную, которая является синонимом другой переменной. Многоуровневые ссылки (то есть, ссылка на ссылку на переменную, как это можно делать, например, в Perl) не поддерживаются. Так что не стоит воспринимать жесткие ссылки серьезнее, чем синонимы.

Чтобы создать жесткую ссылку, нужно использовать оператор **&** (амперсанд). Например:

<?php

$a = 10;

$b = &$a; *// теперь $b — то же самое, что и $a*

echo "b=$b, a=$a"; *// Выводит: "b=10, a=10"*

echo "<br>";

$b = 0; *// на самом деле $a=0*

echo "b=$b, a=$a"; *// Выводит: "b=0, a=0"*

?>

Ссылаться можно не только на переменные, но и на элементы массива (этим жесткие ссылки выгодно отличаются от символических). Например:

<?php

$A=array('a' => 'aaa', 'b' => 'bbb');

$b=&$A['b']; *// теперь $b — то же, что и элемент с индексом 'b' массива*

echo $A['b'],"<br>"; *// Выводит bbb*

$b=0; *// на самом деле $A['b']=0;*

echo $A['b']; *// Выводит 0*

?>

Впрочем, элемент массива, для которого планируется создать символическую ссылку, может и не существовать. Как в следующем случае:

<?php

$A=array('a' => 'aaa', 'b' => 'bbb');

$b=&$A['c']; *// теперь $b — то же, что и элемент с индексом 'c' массива*

echo "Элемент с индексом 'c': (".$A['c'].")";

?>

В результате выполнения рассмотренного скрипта, хотя ссылке **$b** и не было ничего присвоено, в массиве **$A** создастся новый элемент с ключом **c** и значением – пустой строкой (мы можем это определить по результату работы **echo**). То есть, жесткая ссылка на самом деле не может ссылаться на несуществующий объект, а если делается такая попытка, то объект создается.

**Примечание**: если убрать строку, в которой создается жесткая ссылка, то будет выведено сообщение о том, что элемент с ключом **c** не определен в массиве **$A**.

Жесткие ссылки удобно применять при передаче параметров пользовательской функции и возврате значения из нее.

### **Символические ссылки (переменные переменные)**

**Символическая ссылка** – это всего лишь строковая переменная, хранящая имя другой переменной. Чтобы добраться до значения переменной, на которую ссылается символическая ссылка, необходимо применить дополнительный знак **$** перед именем ссылки. Рассмотрим пример:

<?php

$a = 10;

$b = 20;

$c = 30;

$p="a"; *// или $p="b" или $p="c" (присваиваем $p имя другой переменной)*

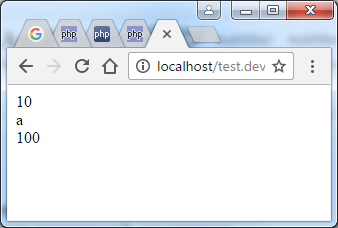
echo $$p,"<br>"; *// выводит переменную, на которую ссылается $p, т. е. $a*

echo $p,"<br>";

$$p=100; *// присваивает $a значение 100*

echo $a;

?>



Мы видим, что для того, чтобы использовать обычную строковую переменную как ссылку, нужно перед ней поставить еще один символ **$**.Это говорит интерпретатору, что надо взять не значение самой **$p**, а значение переменной, имя которой хранится в переменной **$p**.

### **Передача значений по ссылке с использованием функций**

Вы можете передавать переменные в пользовательскую функцию по ссылке, если вы хотите разрешить функции модифицировать свои аргументы. В таком случае, пользовательская функция сможет изменять аргументы.

Синтаксис таков:

<?php  
function foo(&$var)  
{  
     $var++;  
}  
  
$a=5;  
foo($a);

echo $a;  
*// $a здесь равно 6*  
?>

Заметьте, что в вызове функции отсутствует знак ссылки – он есть только в определении функции. Этого достаточно для корректной передачи аргументов по ссылке.

### **Удаление ссылок (сброс ссылок)**

При удалении ссылки, просто разрывается связь имени и содержимого переменной. Это не означает, что содержимое переменной будет разрушено.

Например:

<?php

$a = 1;

$b =&$a;

echo "a=$a",", b=$b";

unset($a);

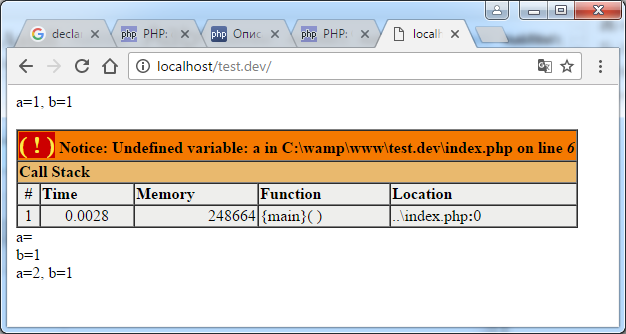
echo "<br>","a=$a";

echo "<br>","b=$b";

$a=2;

echo "<br>","a=$a",", b=$b";

?>



Этот код не сбросит **$b**, а только **$a**.

И все же, жесткая ссылка – не абсолютно точный синоним объекта, на который она ссылается. Дело в том, что оператор [**Unset()**](http://www.php.su/functions/?unset), выполненный для жесткой ссылки, не удаляет объект, на который она ссылается, а всего лишь разрывает связь между ссылкой и объектом.

Итак, жесткая ссылка и переменная (объект), на которую она ссылается, совершенно равноправны, но изменение одной влечет изменение другой

Оператор **Unset()** разрывает связь между объектом и ссылкой, но объект удаляется только тогда, когда на него никто уже не ссылается.

## **Пользовательские функции в PHP**

В любом языке программирования существуют подпрограммы. В языке C они называются функциями, в ассемблере – подпрограммами, а в Pascal существуют даже два вида подпрограмм: процедуры и функции.

**Подпрограмма** – это специальным образом оформленный фрагмент программы, к которому можно обратиться из любого места внутри программы. Подпрограммы существенно упрощают жизнь программистам, улучшая читабельность исходного кода, а также сокращая его, поскольку отдельные фрагменты кода не нужно писать несколько раз

В PHP такими подпрограммами являются пользовательские функции.

Помимо встроенных функций PHP, часто возникает необходимость создания пользовательских функций, выполняющих определенные задачи.

### **Создание пользовательских функций**

Пользовательская функция может быть объявлена в любой части программы (скрипта), до места ее первого использования. И не нужно никакого предварительного объявления, как в других языках программирования, в частности, в C. Преимущества применяемого в PHP подхода в следующем.

Дойдя до определения пользовательской функции, транслятор проверит корректность определения и выполнит трансляцию определения функции во внутреннее представление, но транслировать сам код он не будет. И это правильно – зачем транслировать код, который, возможно, вообще не будет использован. Синтаксис объявления функций следующий:

function Имя (аргумент1[=значение1],...,аргумент1[=значение1])  
{  
тело\_функции  
}

Объявление функции начинается служебным словом **function**, затем следует имя функции, после имени функции – список аргументов в скобках. Тело функции заключается в фигурные скобки и может содержать любое количество операторов.

Требования, предъявляемые к именам функций:

1. имена функций могут содержать русские буквы, но давать функциям имена, состоящие из русских букв не рекомендуется;
2. имена функций не должны содержать пробелов;
3. имя каждой пользовательской функции должно быть уникальным. При этом, необходимо помнить, что регистр при объявлении функций и обращении к ним не учитывается. То есть, например, функции funct() и FUNCT() имеют одинаковые имена;
4. функциям можно давать такие же имена, как и переменным, только без знака $ в начале имен.
5. Типы значений, возвращаемые пользовательскими функциями, могут быть любыми. Для передачи результата работы пользовательских функций в основную программу (скрипт) используется конструкция **return**. Если функция ничего не возвращает, конструкцию **return** не указывают. Конструкция **return** может возвращать все, что угодно, в том числе и массивы.

Приведем примеры использования пользовательских функций:

<?php   
function funct() {  
$number = 777;  
 return $number;  
}  
$a = funct();  
echo $a;  
?>

В рассмотренном примере функция funct возвращает с помощью конструкции **return** число 777. Возвращенное функцией значение присваивается глобальной переменной **$a**, а затем оператор echo выводит значение переменной **$a** в браузер. В результате мы увидим в браузере число 777.

## **Особенности пользовательских функций PHP**

Перечислим особенности пользовательских функций в PHP:

1. Доступны параметры по умолчанию. Есть возможность вызывать одну и ту же функцию с переменным числом параметров;
2. Пользовательские функции могут возвращать любой тип;
3. Область видимости переменных внутри функции является иерархической (древовидной);
4. Есть возможность изменять переменные, переданные в качестве аргумента.

В PHP программисту дана достаточно высокая свобода при создании пользовательских функций. В отличии от языка C++ в пользовательских функциях доступны параметры по умолчанию.

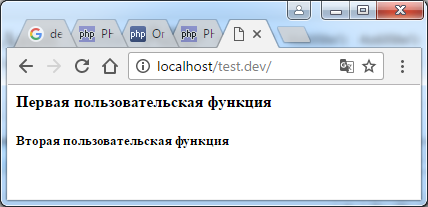
Недостатком пользовательских функций PHP является:

1. Невозможность объявления локальных функций. В PHP вы не можете объявить локальную функцию, как это можно сделать в других языках программирования. Попросту говоря, вы не можете создать функцию внутри другой функции таким образом, чтобы первая (вложенная) функция была видна только во второй функции. В PHP вложенная функция будет доступна всей программе (скрипту), а значит не будет локальной.

Рассмотрим пример:

<?php   
function first\_function() {  
  echo "<h4>Первая пользовательская функция</h4>";  
    function second\_function() {  
     echo "<h5>Вторая пользовательская функция</h5>";  
    }  
}  
first\_function();  
second\_function();  
?>

В рассмотренном примере сценарий выведет:



Значит, обе функции доступны программе. Это говорит о том, что вторая функция не является локальной.

2. Второй недостаток пользовательских функций PHP связан с областью видимости функций. Чтобы разобраться в этой категории, необходимо пояснить, какие переменные являются глобальными, а какие – локальными.

**Глобальные переменные** – это переменные, которые доступны всей программе, включая подпрограммы (функции).

**Локальные переменные** – переменные, определенные внутри подпрограммы (функции). Они доступны только внутри функции, в которой они определены.

Для PHP все объявленные и используемые в функции переменные по умолчанию локальны для функции. То есть, по умолчанию нет возможности изменить значение глобальной переменной в теле функции.

Если вы в теле пользовательской функции будете использовать переменную с именем, идентичным имени глобальной переменной (находящейся вне пользовательской функции), то никакого к отношения глобальной переменной эта локальный переменная иметь не будет. В данной ситуации в пользовательской функции будет создана локальная переменная с именем, идентичным имени глобальной переменной, но доступна данная локальная переменная будет только внутри этой пользовательской функции.

Поясним данный факт на конкретном примере:

<?php   
$a = 100; */\* глобальная область видимости \*/*  
  
function funct() {  
$a = 70; */\* ссылка на переменную локальной области видимости \*/* echo "<h4>$a</h4>";  
}  
funct();  
echo "<h2>$a</h2>";  
?>

Сценарий выведет сперва 70, а затем 100.

При выходе из пользовательской функции, в которой была объявлена локальная переменная, эта переменная и ее значение уничтожаются.

Основное достоинство локальных переменных – отсутствие непредвиденных побочных эффектов, связанных со случайной или намеренной модификацией глобальной переменной. Рассмотрим еще пример:

<?php  
$a = 1; */\* глобальная область видимости \*/*  
function Test()  
{   
     echo $a; */\* ссылка на переменную локальной области видимости \*/*   
}   
Test();  
?>

Этот скрипт не сгенерирует никакого вывода, поскольку выражение **echo** указывает на локальную переменную **$a**, а в пределах локальной области видимости ей не не было присвоено значение.

Подход к области видимости в PHP отличается от языка C в том, что глобальные переменные в C автоматически доступны функциям, если только они не были перезаписаны локальным определением.

В PHP существует специальная инструкция **global**, позволяющая пользовательской функции работать с глобальными переменными. Рассмотрим данный принцип на конкретных примерах:

<?php  
$a = 1;  
$b = 2;  
  
function Sum()  
{  
     global $a, $b;  
  
     $b = $a + $b;  
}   
  
Sum();  
echo $b;  
?>

Вышеприведенный скрипт выведет "**3**". После определения **$a** и **$b** внутри функции как **global** все ссылки на любую из этих переменных будут указывать на их глобальную версию.

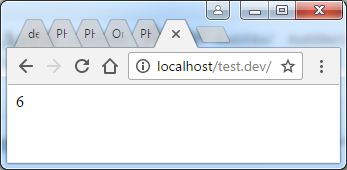
Не существует никаких ограничений на количество глобальных переменных, которые могут обрабатываться пользовательскими функциями.

Второй способ доступа к переменным глобальной области видимости – использование специального, определяемого PHP массива **$GLOBALS**.

Предыдущий пример может быть переписан так:

Использование $GLOBALS вместо global:

<?php  
$a = 1;  
$b = 5;  
  
function Sum()  
{  
     $GLOBALS["c"] = $GLOBALS["a"] + $GLOBALS["b"];  
}   
  
Sum();  
echo $c;  
?>



**$GLOBALS** – это ассоциативный массив, ключом которого является имя, а значением – содержимое глобальной переменной. Обратите внимание, что $GLOBALS существует в любой области видимости.

## **Передача аргументов пользовательским функциям**

При объявлении функции можно указать список параметров, которые могут передаваться функции, например:

<?php  
function funct($a, $b, /\* ..., \*/ $z) { ... };  
?>

При вызове функции funct() нужно указать все передаваемые параметры, поскольку они являются обязательными. В PHP пользовательские функции могут обладать необязательными параметрами или параметрами по умолчанию, но об этом позже.

## **Передача аргументов по ссылке**

Согласно сложившимся традициям, во всех языках программирования есть два вида аргументов функций:

* параметры-значения;
* параметры-переменные.

Функции не могут изменить параметр-значение, то есть он доступен функции «только для чтения» – она может его использовать, но не более. В качестве параметра-значения необязательно указывать переменную, можно указать само значение, отсюда название – параметр-значение.

По умолчанию аргументы в функцию передаются по значению (это означает, что если вы измените значение аргумента внутри функции, то вне ее значение все равно останется прежним). Приведем пример:

<?php

function funct($string)

{

echo "<h3>Параметр = $string </h3>";

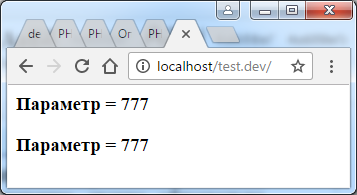
}

$str = 777;

funct(777);

funct($str);

?>



В отличие от параметров-значений, параметры-переменные могут быть изменены в процессе работы функции. Тут уже нельзя передавать значение, нужно обязательно передать переменную. В PHP для объявления параметров-переменных используется механизм передачи переменной по ссылке.

Если вы хотите разрешить функции модифицировать свои аргументы, вы должны передавать их по ссылке.

Если вы хотите, чтобы аргумент всегда передавался по ссылке, вы должны указать амперсанд (&) перед именем аргумента в описании функции:

<?php

function funct(&$string)

{

$string .= ' А эта внутри.';

}

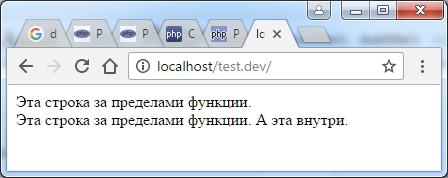
$str = 'Эта строка за пределами функции. ';

echo $str,"<br>";

funct($str);

echo $str;

?>



## **Параметры по умолчанию**

При программировании часто возникает необходимость создания функции с переменным числом параметров. Тому есть две причины:

* Параметров слишком много. При этом нет смысла каждый раз указывать все параметры;
* Функции должны возвращать значения разных типов в зависимости от набора параметров.

В PHP функции могут возвращать любые значения в зависимости от переданных им параметров.

<?php

function makecup($type = "Чая")

{

return "Сделайте чашечку $type";

}

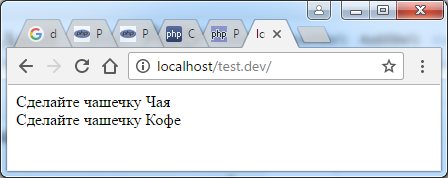
echo makecup();

echo "<br>";

echo makecup("Кофе");

?>

Результат работы приведенного скрипта будет таким:



PHP также позволяет использовать массивы и специальный тип NULL в качестве значений по умолчанию. Значение по умолчанию должно быть константным выражением.

Обратите внимание, что все аргументы, для которых установлены значения по умолчанию, должны находится правее аргументов, для которых значения по умолчанию не заданы, в противном случае ваш код может работать не так, как вы этого ожидаете. Рассмотрим следующий пример:

<?php

function makecup($type = "чая", $cond)

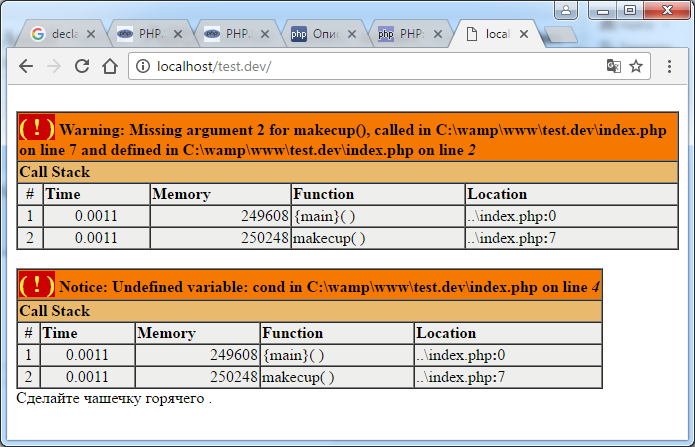
{

return "Сделайте чашечку $type $cond.";

}

echo makecup("горячего"); *// Не будет работать так, как мы могли бы ожидать*

?>



<?php

function makecup($cond, $type = "чая")

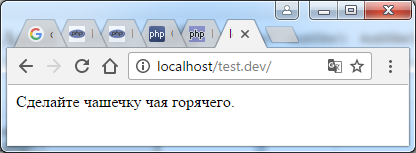
{

return "Сделайте чашечку $type $cond.";

}

echo makecup("горячего");

?>



Начиная с PHP 5, значения по умолчанию могут быть переданы по ссылке!

## **Переменное число аргументов в функциях**

Иногда мы точно не знаем, сколько параметров будет передано нашей функции. Специально для такого случая разработчики PHP предусмотрели возможность использования переменного числа аргументов

Реализация этой возможности достаточно прозрачна и заключается в использовании функций **func\_num\_args()**, **func\_get\_arg()** и **func\_get\_args()**.

Рассмотрим возможности рассмотренных стандартных функций:

Стандартная функция **func\_num\_args()** возвращает количество аргументов, переданных пользовательской функции:

<?php  
function funct()   
{  
     $numargums = func\_num\_args();  
     echo "Количество аргументов : $numargums\n";  
}   
  
funct(1, 2, 3);    *// Скрипт выведет 'Количество аргументов: 3'*  
?>

Стандартная функция **func\_get\_arg()** возвращает элемент из списка переданных пользовательской функции аргументов:

<?php  
function funct()   
{  
      $numargs = func\_num\_args();  
      echo "Количество аргументов: $numargs<br>";  
      if ($numargs >= 2) {  
      echo "Второй аргумент : ".func\_get\_arg(1)."<br>";  
      }  
}   
funct(1, 2, 3);  
?>

Стандартная функция **func\_get\_args()** возвращает массив аргументов, переданных пользовательской функции:

<?php  
function funct()   
{  
     $numargs = func\_num\_args();  
     echo "Количество аргументов : $numargs<br>";  
     if ($numargs >= 2) {  
         echo "Второй аргумент: " . func\_get\_arg(1) . "<br>";  
     }  
     $arg\_list = func\_get\_args();  
     for ($i = 0; $i < $numargs; $i++) {  
         echo "Аргумент $i is: " . $arg\_list[$i] . "<br>";  
     }  
}   
funct(1, 2, 3);  
?>

Обратите внимание, что при объявлении пользовательских функций в скобках мы ничего не пишем, то есть, как будто мы не передаем аргументы.

## **Cтатические переменные**

Помимо локальных и глобальных переменных, в PHP существует еще один тип переменных – **статические переменные**.

Если в теле пользовательской функции объявлена статическая переменная, то компилятор не будет ее удалять после завершения работы функции. Пример работы пользовательской функции, содержащей статические переменные:

<?php  
   function funct()  
   {  
      static $a;  
      $a++;  
      echo "$a";  
   }  
   for ($i = 0; $i++<10;) funct();  
?>

Данный сценарий выводит строку:

1 2 3 4 5 6 7 8 9 10

Если мы удалим инструкцию static, будет выведена иная строка.

<?php

function funct()

{

$a++;

echo "$a";

}

for ($i = 0; $i++<10;) funct();

?>

1 1 1 1 1 1 1 1 1 1

Это связано с тем, что переменная **$a** будет удаляться при завершении работы функции и обнуляться при каждом ее вызове. Переменная **$a** инкрементируется сразу после обнуления, а только потом выводится.

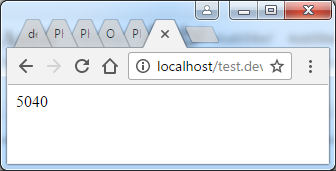
## **Рекурсивные функции**

**Рекурсивные функции** – это функции, вызывающие самих себя. Такой вызов называется рекурсивным. Рекурсия бывает:

* прямая;
* непрямая.

Рассмотрим пример рекурсивной функции (прямой), используемой при вычислении факториала **x!** :

<?php  
function factorial($x) {  
if ($x === 0) return 1;  
else return $x\*factorial($x-1);  
}  
echo factorial(7);  
?>



В рассмотренном примере пользовательская функция **factorial()** вызывает сама себя, что является прямой рекурсией.

Непрямая рекурсия возникает, когда первая функция вызывает вторую, а вторая – первую.

При создании рекурсивных функций необходимо соблюдать осторожность, стараясь избегать зацикливания.

## **Условно определяемые функции**

PHP позволяет, в зависимости от определенных факторов, одной и той же функции выполнять различные действия. Рассмотрим пример:

<?php  
$phpver = phpversion();  
if ($phpver[0] === "5")  
{  
function getversion() { return "Вы используете PHP5"; }  
}  
if ($phpver[0] === "4")  
{  
function getversion() { return "Вы используете PHP4"; }  
}  
if ($phpver[0] === "3")  
{  
function getversion() { return "Вы используете PHP3"; }  
}  
echo @getversion();  
?>

Рассмотренный скрипт выводит версию используемого интерпретатора PHP. Одна и та же функция**getversion()** может возвращать различный результат, в зависимости от значения переменной **$phpver**