Лабораторная работа 2

Типы

Переменные бывают разных типов и иногда нам нужно узнать какого типа переменная, поменять её тип чтобы сравнить с переменной другого типа или сделать с ней что нибудь полезное.

Тип переменной по умолчанию выбирается автоматически при присваивании.

Так же автоматически приводятся типы переменных при операциях с ними, если это необходимо. Так же как мы ранее заметили, при делении 5 на 2,5 результатам будет дробь, так как интерпретатор приводит обе переменных к одному типу и потом производит деление.

Узнать какой тип у переменной можно при помощи функции **gettype();**

Функция это набор действий. Она принимает аргументы, делает с ними некие операции и возвращает результат. К примеру Приготовить пиццу это функция принимающая параметры - тесто, яйца, сыр, кетчуп, анчоусы, плиту, и возвращает Пиццу.

Возвращаемое значение у функций может быть любого типа данных.

У функции getttype() возвращаемый результат это строка содержащие название типа.

Если вы попробуете код echo gettype(512); то получите в ответ строку "Integer".

Можно узнать является ли тем или иным типом данных переменная при помощи **is_float(), is_int(), is_string(), is_object(), is_array().**

Эти функции принимают один аргумент и возвращают булев тип (true или false).

Изменить тип переменной можно несколькими функциями

intval() - возвращает аргумент в виде целого числа integer

floatval() - возвращает аргумент в виде дробного числа float

strval() - возвращает аргумент в виде строки string

settype() - превращает первый аргумент в указанный во втором аргументе тип settype($a,'integer') превратит $a в целое число.

Заметьте что эта функция не возвращает нового значения, а изменяет переданный ей параметр.

Допустимыми значениями параметра type являются:

* **"boolean"**
* **"integer"** (или **"int"**)
* **"float"**
* **"string"**
* **"array"**
* **"object"**
* **"null"**

Операции с переменными

Арифметические операторы

-$a	**Отрицание** Смена знака $a.

$a + $b Сложение Сумма $a и $b.
$a - $b Вычитание Разность $a и $b.
$a * $b Умножение Произведение $a и $b.
$a / $b Деление Частное от деления $a на $b.
$a % $b Деление по модулю Целочисленный остаток от деления $a на $b.

Операторы инкремента и декремента

++$a Префиксный инкремент Увеличивает $a на и возвращает значение $a.
$a++ Постфиксный инкремент Возвращает значение $a, а затем увеличивает $a на 1.
--$a Префиксный декремент Уменьшает $a на 1 и возвращает значение $a.
$a-- Постфиксный декремент Возвращает значение $a, а затем уменьшает $a на 1.

Таким образом $a = 5; echo $a++ ; Сначала выведет 5, а потом увеличит 5 на 1.

Логические операторы

$a and $b Логическое 'и' TRUE если и $a, и $b TRUE.
$a or $b Логическое 'или' TRUE если или $a, или $b TRUE.
$a xor $b Исключающее 'или' TRUE если $a, или $b TRUE, но не оба.
! $a Отрицание TRUE если $a не TRUE.
$a && $b Логическое 'и' TRUE если и $a, и $b TRUE.
$a || $b Логическое 'или' TRUE если или $a, или $b TRUE.

Строковые операторы

В PHP есть два оператора для работы со строками. Первый - оператор конкатенации ('.'), который возвращает объединение левого и правого аргумента в одну строку. Второй - оператор присвоения вместе с конкатенацией.

```
<?PHP
$a = "Hello ";
$b = $a . "World!"; // $b содержит строку "Hello World!"
$a = "Hello ";
$a .= "World!";     // $a содержит строку "Hello World!"
?>
```

Побитовые операторы

Побитовые операторы позволяют устанавливать конкретные биты в 0 или 1 для целочисленных значений. В случае если и левый, и правый операнды строки, побитовые операции будут работать с их ASCII-представлениями.

$a & $b Побитовое 'и' Устанавливаются только те биты, которые установлены и в $a, и в $b.

$a | $b Побитовое 'или' Устанавливаются те биты, которые установлены либо в $a, либо в $b.
$a ^ $b Исключающее'или' Устанавливаются только те биты, которые установлены либо только в $a, либо только в $b
~ $a Отрицание Устанавливаются те биты, которые в $a не установлены, и наоборот.
$a << $b Сдвиг влево Все биты переменной $a сдвигаются на $b позиций влево (каждая позиция подразумевает 'умножение на 2')
$a >> $b Сдвиг вправо Все биты переменной $a сдвигаются на $b позиций вправо (каждая позиция подразумевает 'деление на 2')

Операторы сравнения

$a == $b Равно TRUE если $a равно $b.
$a === $b Тождественно равно TRUE если $a равно $b и имеет тот же тип.
$a != $b Не равно TRUE если $a не равно $b.
$a <> $b Не равно TRUE если $a не равно $b.
$a !== $b Тождественно не равно TRUE если $a не равно $b или они разных типов
$a < $b Меньше TRUE если $a строго меньше $b.
$a > $b Больше TRUE если $a строго больше $b.
$a <= $b Меньше или равно TRUE если $a is меньше или равно $b.
$a >= $b Больше или равно TRUE если $a больше или равно $b.

Операторы работы с массивами

$a + $b Объединение Объединение массива $a и массива $b.
$a == $b Равно TRUE в случае, если $a и $b содержат одни и те же элементы.
$a === $b Тождественно равно TRUE в случае, если $a и $b содержат одни и те же элементы в том же самом порядке.
$a != $b Не равно TRUE если массив $a не равен массиву $b.
$a <> $b Не равно TRUE если массив $a не равен массиву $b.
$a !== $b Тождественно не равно TRUE если массив $a не равен тождественно массиву $b.

Управляющие конструкции.

Так как любая программа создается человеком, то (как правило), в нее вкладывается некоторый смысл. То есть логика.

А для управления логикой существуют управляющие конструкции. Ведь, по сути, что значит исполнение программы? Это - пошаговое исполнение, инструкция за инструкцией, кода, написанного программистом.

Для управления ходом этого самого исполнения нам пригождаются конструкции. Отмечу, что они такие не только в PHP, но и почти везде, разница лишь в синтаксисе.

Самым первым рассмотрим **ветвление**. В самом деле - ведь очень

часто нам хочется в одном случае заставить программу "выполнить" что-то одно, а в другом - нечто другое.

Вот например, есть строка, содержащая имя. Как понять понять, мое ли оно:

```
<?PHP
//мы предположим, что в ходе выполнения программы мы откуда-то взяли переменную $sName
if($sName=="EuGen")
{
  echo("Да, это я");
}
else
{
  echo("Наверное, это Valenok");
}
?>
```

Немного прокомментируем - для понимания, что такое переменная, что значит "==" - смотрите предыдущий урок. Как мы видим, ветвление - это оператор **if**.

Он просто проверяет условие. Условие – это **выражение** логического типа. Логического - значит, принимающего одно из значений - ложь или истина. Наши компьютеры столь совершенны, что способны отличить ложь от истины, и, значит, выполнить нужный участок кода.

Если условие истинно - исполняется блок внутри if. Если же нет - то можно дописать **необязательный** блок внутри **else**. Он выполнится, если условие ложно. Но можно его и не писать, в таком случае программа продолжит свое исполнение на первом следующем за **if** операторе. Здесь важно понимать, что на первом не **внутри** блока, а **после** блока **if**.

Очень важно понимать, что условный оператор - такой же оператор как и все, а потому можно считать блоки кода внутри него - некой неделимой частью.

Однако условный оператор умеет больше. Мы можем задать не одно, а сразу несколько условий для того, чтобы управлять исполнением сразу нескольких участков

```
<?PHP
//$sName я опять же взял "откуда-то раньше"
if($sName=="EuGen")
{
  echo("Да, это я");
}
elseif($sName=="Valenok");
{
  echo("Теперь я точно уверен, это Valenok");
}
```

```
elseif($sName=="Champion");
{
  echo("Это тоже наш автор, Champion");
}
else
{
  echo("Я запутался..");
}
?>
```

Как видим, можно писать этих блоков сколько угодно.

В PHP существует еще и очень полезный "быстрый" способ присвоить значение переменной, используя ветвление. Это не оператор if, но выглядит очень похоже.

Вот как это можно сделать:

```
<?PHP
$result=$sName=="EuGen"?"Я":"Не я";
?>
```

Это может выглядеть необычно, но на самом деле все просто. Мы присваиваем переменной $result значение выражения, которое вычисляется в момент исполнения оператора присваивания. В приведенном выражении участвует еще одна переменная, $sName, она-то и будет управлять тем, что попадет в конечном итоге в значение $result.

При исполнении присваивания проверится значение выражения **$sName=="EuGen"**. Если оно - истинно, то в переменную попадет то, что следует после знака "?". Иначе - то, что следует после знака ":".

А теперь немного подумайте и ответьте, что попадет в $result в результате:

```
<?PHP
$sName="Who is it?";
$result=$sName=="EuGen"?"Я":($sName=="Valenok"?"Это Valenok":"Кто-
то другой");
?>
```

Итак, условный оператор позволяет управлять выполнением программы, проверяя одно или несколько логических выражений - условий. При этом в зависимости от истинности или ложности условий будут выполняться соответствующие участки кода.

Рассчмотрим **циклы**

Что такое цикл? Представьте себе, что вам нужно найти сумму первых 20 натуральных чисел. Вот как это будет выглядеть, если использовать то, что мы уже знаем:

```
<?PHP
$iSum=1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20;
//кошмар!
?>
```

Как видим, это очень неудобно. Ведь, предположим, что мы хотим найти сумму первых 50000 натуральных чисел, и тогда ...

Для решения нашей задачи очень подойдет цикл. Цикл - это тоже оператор, и это следует помнить. Цикл позволяет выполнить заданное количество раз один и тот же участок кода.

Как задается это самое "количество раз"? По-разному. Строго говоря, есть разные способы организовать циклы, но по сути все они отличаются лишь способом задания этого самого "количества раз", которое выполнится цикл.

Рассмотрим самый простой из циклов – цикл **for**

Он позволяет выполнить участок кода фиксированное число раз:

```
<?PHP
$iSum=0;
for($i=1; $i<=20; $i++)
{
  $iSum+=$i;
}
?>
```

Вот так это будет выглядеть с использованием цикла.

Терминология: блок кода внутри цикла называется **телом цикла**. Переменная-счетчик называется **итератором цикла**, выполнение блока кода циклом - **итерацией цикла**.

Цикл **for** выполняется, пока истинно выражение, стоящее **вторым** в списке выражений в скобках. Но чтобы начать исполнение, нам нужно с чем-то начать. Для этого предназначено **первое** выражение в скобках **for**. Выражение, стоящее **третьим** будет вычисляться **при каждой итерации цикла**.

В данном примере:

- при входе в цикл создастся переменная $i, и ей присвоится значение 1
- цикл будет выполняться, пока $i не станет равным 20
- при каждом шаге цикла $i будет увеличиваться на 1.

А при выполнении цикла будет проводиться добавление к $iSum значения, которое в момент конкретной итерации находится в $i. Таким образом, мы пройдем все целые числа от 1 до 20 и каждый раз будем добавлять из к $iSum.

Таким образом, если мы захотим сложить первые 50000 натуральных чисел, нам нужно просто изменить условие, до которого цикл будет выполняться: for($i=1; $i<=50000; $i++)

Теперь же подумайте, что попадет в $iSum в результате:

```
<?PHP
$iSum=0;
for($i=0;$i++<=20;)
{
  $iSum+=$i;
```

```
}
?>
```

Подскажу - не забывайте, выражения могут быть пустыми. Бывает так, что необходимо менять сразу несколько выражений для каждой итерации цикла. Их можно записывать через знак запятой: ",".

Например:

```
<?PHP
$sRes="";
for($i=0,$j=10;$i!=$j;$i++,$j--)
{
  $sRes.="$i"."$j";
}
//в $sRes попадет 01019283746 - подумайте почему
?>
```

Ничто так же не мешает нам организовать цикл внутри цикла. Например, таблица умножения от 1 до 9 организуется в два счета (точнее, в 2 цикла):

```
<?PHP
for($i=1;$i<=9;$i++)
{
  for($j=1;$j<=9;$j++)
  {
    echo($i*$j." ");
  }
  echo("\n");
}
//подумайте, зачем я вывожу \n в конце 1-
го цикла, и, кстати, вспомните, что это такое - "\n"
?>
<?PHP
$i=1;
$iSum=0;
while($i<=20)
{
  $iSum+=$i;
  $i++;
}
?>
```

Цикл будет выполняться до тех пор, пока истинно условие, стоящее в скобках. При этом, если оно в момент входа в цикл было ложно, то **цикл не выполнится ни разу**. По факту, мы переписали вычисление нашей суммы с использованием цикла while. Цикл while как правило используют, если нужен цикл, для которого заранее неизвестно число повторений.

У цикла while есть "собрат" – цикл **do**. Он выглядит так:

```
<?PHP
$i=1;
$iSum=0;
do
{
  $iSum+=$i;
  $i++;
}
while($i<=20);
?>
```

Этот цикл также "работает" до тех пор, пока истинно условие, стоящее внутри while, но, в отличии от самого while, он **всегда выполнится хотя бы 1 раз**, даже если исходное выражение, стоящее в условии было сразу ложно.

Так как я рассказываю про циклы, будет важно знать, что существует специальный оператор, прерывающий выполнение цикла "досрочно", то есть до наступления состояния условия, при котором цикл завершался бы, если бы "все шло как обычно". Это – оператор **break**. Он позволяет выйти из цикла немедленно и сразу же перейти к оператору, идущему сразу после цикла.

Например:

```
<?PHP
$i=1;
$iSum=0;
while($i<=200000)
{
  $iSum+=$i;
  if($i==100)
  {
    break;
  }
  $i++;
}
?>
```

Это выполнит сложение лишь первых 100 чисел, а не первых 200000 (несмотря на условие цикла), так как когда $i достигнет 100, произойдет выполнение break, и, значит, выход из цикла.

break "умеет" выходить с учетом вложенности циклов. Иначе говоря, можно задавать, куда именно мы хотим выйти. Вернемся к примеру с таблицей умножения, но изменим его:

```
<?PHP
for($i=1;$i<=9;$i++)
{
  for($j=1;$j<=10000;$j++)
  {
```

```
    echo($i*$j." ");
    if($j==9)
    {
      break(2);
    }
  }
}
?>
```

break принимает параметр - вложенность. В данном примере мы сразу же выйдем из обоих циклов, так как их всего 2. По умолчанию применяется значение 1, то есть break(1) равносильно break

Кроме break существует еще один управляющий циклом оператор - **continue**. Он позволяет сразу же переходить к следующей итерации цикла, не выполняя в текущей ничего, что стоит **после** этого оператора:

```
<?PHP
for($i=1;$i<=9;$i++)
{
  for($j=1;$j<=11;$j++)
  {
    if($j>9)
    {
      continue;
    }
    echo($i*$j." ");
  }
  echo("\n");
}
?>
```

Понятно, что особого смысла размещать **continue** в конце блока кода, исполняемого внутри цикла, нет.

Теперь перейдем к еще одной конструкции, очень часто применяемой в php. Это – цикл **foreach**. Он исключительно удобен для обработки массивов. Выглядит он очень просто. Для примера код, который просто выводит все элементы обычного массива:

```
<?PHP
//$rgData что-либо содержит, это массив
foreach($rgData as $item)
{
  echo($item."\n");
}
?>
```

Как видим, этот цикл пробегается по всему массиву. А текущий элемент массива доступен в переменной **$item**.

Но этот цикл умеет больше. Очень часто нам нужен не только сам

элемент, или, если быть точным, значение элемента, но и его ключ. В следующем примере вывод делается уже с ключами:

```
<?PHP
//$rgData что-либо содержит, это массив
foreach($rgData as $key=>$value)
{
  echo("Элемент массива '".$key."' равен: ".$value."\n");
}
?>
```

Конструкция лишь немного усложнилась - в скобках мы указываем ключ=>значение. И соответствующие ключ и значение текущего элемента массива в теле цикла доступны как переменные **$key** и **$value**.

Небольшое отступление - те, кто разбирается в математике чуть глубже, могут легко мне возразить, и написать простенькую программу, которая и без цикла вычислит сумму первых 50000 натуральных чисел:

```
<?PHP
$iCount=50000;
$result=$iCount*(1+$iCount)/2;
//..вот и все, при чем тут циклы?
?>
```

Перейдем к еще одной управляющей конструкции php. А именно - к оператору **выбора**. Этот оператор позволяет делать более сложные ветвления, чем if, более простым способом. На самом деле, при помощи if можно полностью заменить этот оператор, но знать про него полезно.

Итак, этот оператор называется **switch**. В дословном переводе - "переключатель", так оно и есть. В лучших традициях, сразу же перейдем к примерам. Пусть мы хотим сделать следующую вещь: вывести день недели по его порядковому номеру.

Сначала приведу решение, а потом поясню:

```
<?PHP
//$iDayNum получено ранее
switch($iDayNum)
{
  case 1:
    echo("Понедельник");
    break;
  case 2:
    echo("Вторник");
    break;
  case 3:
    echo("Среда");
    break;
  case 4:
    echo("Четверг");
```

```
    break;
  case 5:
    echo("Пятница");
    break;
  case 6:
    echo("Суббота");
    break;
  case 7:
    echo("Воскресенье");
    break;
  default:
    echo("А у нас в неделе только 7 дней...");
}
?>
```

При входе в оператор switch, сначала вычисляется выражение, стоящее в скобках. Затем перебираются все значения, указанные при помощи case. В данном примере мы видим 7 разных значений. Соответственно, когда значение, которое имеет вычисленное в скобках выражение, найдено у какого-либо case, начинается исполнение участка когда внутри case. И вот здесь важно понимать, что означает "внутри case". Когда соответствующее значение найдено, произойдет исполнение **всего, что есть дальше того case, которому соответствует значение**. И это значит, что если после этого case есть другие, **они тоже будут выполнены**. Это может быть не то, что нам нужно.

И здесь нам на помощь приходит знакомый уже нам оператор break. Он позволяет выйти из оператора switch, и, поставив его после кода, записанного внутри каждого case, мы обеспечим себе исполнение только одного участка кода для каждого значения.

Кроме этого, у оператора switch может быть необязательная часть **default**. Код в ней выполнится, если значение выражения не совпало ни с одним из значений в case.

Это очень похоже на часть **else** для оператора **if .. elseif .. else**

Но не стоит забывать, что switch и if - не одно и то же. Иногда вам может пригодиться возможность switch исполнять весь участок кода, включая другие case, после того как найдено совпадение.

Вопросы

0. Когда наличие else обязательно в операторе if?
1. Что станет результатом исполнения:

```
<?PHP
$iData=5;
if($iData%5)
{
  echo($iData>0?"case 1":"case 0");
}
```

```
else
{
  echo($iData<0?"case 3":"case 2");
}
?>
```

2. Перепишите код с использованием оператора for:

```
<?PHP
$iData=1;
while($iData%1024)
{
  echo($iData%1024);
  $iData*=2;
}
?>
```

3. Что будет результатом выполнения кода:

```
<?PHP
$i=1;
do
{
  echo($i);
}
while($i<10);
?>
```

4. Предположим у нас есть 3 цикла, вложенных друг в друга. Какой оператор позволит внутри итерации самого внутреннего из них выйти сразу из всех циклов?

5. Зачем нужен break при использовании switch?

6. Предположим есть ассоциативный массив. Напишите программу, которая поменяет местами ключи и значения этого массива.