**Переменные, специальные символы.**

В PL/SQL действует соглашение, что все символы приводятся к верхнему регистру по этому объявления типа:

a VARCHAR2(5);

A VARCHAR2(5);

Одинаковы. Так же, естественно, запрещено использовать зарезервированные имена встроенных функций и пакетов. Каждый законченный оператор обрамляется символом ";". Вообще, если говорить прямо, когда, вы начёнте работать с PL/SQL, уже на первом этапе чётко видно, что сам PL/SQL очень похож на язык Pascal. Итак, давайте разберемся для начала со всеми специальными символами.

Типы специальных символов PL/SQL:

1. Арифметические операторы:

|  |  |
| --- | --- |
| **+** | Сложение и унарный плюс |
| **-** | Вычитание и унарный минус |
| **\*** | Умножение |
| **/** | Деление |
| **\*\*** | Возведение в степень |

1. Операторы отношения (используются в логических выражениях):

|  |  |
| --- | --- |
| **=** | Равенство |
| **<** | Меньше |
| **>** | Больше |
| **<>** | Не равно |
| **!=** | Не равно (альтернатива) |
| **~=** | Не равно (альтернатива) |
| **^=** | Не равно (альтернатива) |
| **<=** | Меньше или равно |
| **>=** | Больше или равно |

1. Выражение и списки (используются в операторах, объявлениях типов данных, объявлениях списков параметров, ссылках на переменные и таблицы):

|  |  |
| --- | --- |
| **:=** | Присвоение |
| **(** | Начало списка или подвыражения |
| **)** | Конец списка или подвыражения |
| **,** | Отдельные элементы списка (как в списке параметров) |
| **..** | Оператор диапазона используется в операторах FOR-IN |
| **||** | Конкатенация строк |
| **=>** | Ассоциация (используется в списке параметров) |
| **;** | Конец выражения |
| **%** | Атрибут курсора или типа объекта |
| **.** | Спецификация объекта |
| **@** | Индикатор удаленной базы данных |
| **'** | Начало/конец строки символов |
| **:** | Индикатор внешней переменной |
| **&** | Индикатор связанной переменной |

1. Комментарии и метки

|  |  |
| --- | --- |
| **--** | Комментарий в одной строке |
| **/\*** | Начало многострочного комментария |
| **\*/** | Конец многострочного комментария |
| **>>** | Начало метки |
| **<<** | Конец метки |

Как видите очень много символов обозначающих одно и тоже, к примеру не равно, хотя **!=**  вполне достаточно. Я например использую **<>** (как в SQL). Но вообще можно использовать любой символ. Вложенные комментарии не допускаются. Проще использовать /\* \*/ как в языке C**.** А теперь про самое интересное – блоки PL/SQL.

Блоки PL/SQL, могут быть, "именованными" и "не именованными". Блок PL/SQL является фундаментальной программной конструкцией. Программирование модулями позволяет разрабатывать легко читаемый код и программировать сверху вниз. Неименованный блок PL/SQL имеет три раздела - Declaration (объявления), Body (тело) и, как правило, Exception (исключения).

Стандартная конструкция неименованного блока:

[DECLARE

-- объявления]

BEGIN

-- выполняемый код

[EXCEPTION

-- обработка исключений]

END;

Блоки могут иметь имя и быть вложенными друг в друга

PROCEDURE calc\_totals

IS

year\_total NUMBER;

BEGIN

year\_total := 0;

/\* Начало вложенного блока \*/

DECLARE

month\_total NUMBER;

BEGIN

month\_total := year\_total / 12;

END set\_month\_total;

/\* Конец вложенного блока \*/

END;

Область видимости

CREATE OR REPLACE PACKAGE scope\_demo

IS

g\_global NUMBER;

PROCEDURE set\_global (number\_in IN NUMBER);

END scope\_demo;

/

CREATE OR REPLACE PACKAGE BODY scope\_demo

IS

PROCEDURE set\_global (number\_in IN NUMBER)

IS

l\_salary NUMBER := 10000;

l\_count PLS\_INTEGER;

BEGIN <<local\_block>>

DECLARE

l\_inner NUMBER;

BEGIN

SELECT COUNT (\*)

INTO l\_count

FROM employees

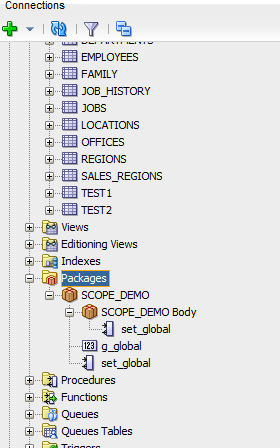
WHERE department\_id = l\_inner AND salary > l\_salary;

END local\_block;

g\_global := number\_in;

END set\_global;

END scope\_demo;



Переменная scope\_demo.g\_global может использоваться в любом блоке любой схемы, обладающем привилегией EXECUTE для scope\_demo. Переменная l\_salary может использоваться только в процедуре set\_global. Переменная l\_inner может использоваться только в локальном или вложенном блоке; обратите внимание на использование метки local\_block для присваивания имени вложенному блоку.

PACKAGE BODY scope\_demo

IS

PROCEDURE set\_global (number\_in IN NUMBER)

IS

l\_salary NUMBER := 10000;

l\_count PLS\_INTEGER;

BEGIN

<<local\_block>>

DECLARE

l\_inner PLS\_INTEGER;

BEGIN

SELECT COUNT (\*)

INTO **set\_global.l\_count**

FROM employees e

WHERE **e.department\_id = local\_block.l\_inner**

AND **e.salary > set\_global.l\_salary;**

END local\_block;

**scope\_demo.g\_global := set\_global.number\_in;**

END set\_global;

END scope\_demo;

Удобство чтения

Практически любая команда SQL, встроенная в программу PL/SQL, содержит ссылки на столбцы и переменные. В небольших, простых командах SQL различать эти ссылки относительно просто. Однако во многих приложениях используются очень длинные, исключительно сложные команды SQL с десятками и даже сотнями ссылок на столбцы и переменные. Без уточнения ссылок вам будет намного сложнее различать переменные и столбцы. С уточнениями сразу видно, к чему относится та или иная ссылка.

«Один момент… Мы используем четко определенные схемы назначения имен, при помощи которых мы различаем строки и столбцы. Имена всех локальных переменных начинаются с l\_, поэтому мы сразу видим, что идентификатор представляет локальную переменную». Да, все правильно; все мы должны иметь (и соблюдать) правила назначения имен, чтобы имена идентификаторов содержали дополнительную информацию о них (что это — параметр, переменная? К какому типу данных она относится?). Безусловно, правила назначения имен полезны, но они еще не гарантируют, что компилятор PL/SQL всегда будет интерпретировать ваши идентификаторы именно так, как вы задумали.

Предотвращение ошибок

Если не уточнять ссылки на переменные PL/SQL во встроенных командах SQL, код, который правильно работает сегодня, может внезапно утратить работоспособность в будущем. И разработчику будет очень трудно понять, что же пошло не так. Вернемся к встроенной команде SQL без уточнения ссылок:

SELECT COUNT (\*)

INTO l\_count

FROM employees

WHERE department\_id = l\_inner AND salary > l\_salary;

Сегодня идентификатор l\_salary однозначно представляет переменную l\_salary, объявленную в процедуре set\_global. Я тестирую свою программу — она работает! Программа поставляется клиентам, все довольны. А через два года пользователи просят своего администратора базы данных добавить в таблицу employees столбец, которому по случайности присваивается имя «l\_salary».

Видите проблему? Во встроенной команде SQL база данных Oracle всегда начинает поиск соответствия для неуточненных идентификаторов со столбцов таблиц. Если найти столбец с указанным именем не удалось, Oracle переходит к поиску среди переменных PL/SQL в области действия. После добавления в таблицу employee столбца l\_salary моей неуточненной ссылке l\_salary в команде SELECT ставится в соответствие не переменная PL/SQL, а столбец таблицы.

Результат? Пакет scope\_demo по-прежнему компилируется без ошибок, но секция WHERE запроса ведет себя не так, как ожидалось. База данных не использует значение переменной l\_salary, а сравнивает значение столбца salary в строке таблицы employees со значением столбца l\_salary той же строки. Отыскать подобную ошибку бывает очень непросто!

Не полагайтесь только на правила назначения имен для предотвращения «коллизий» между идентификаторами; уточняйте ссылки на все имена столбцов и переменных во встроенных командах SQL. Это существенно снизит риск непредсказуемого поведения программ в будущем при возможных модификациях таблиц.

Видимость

Важным свойством переменной, связанным с областью ее действия, является видимость. Данное свойство определяет, можно ли обращаться к переменной только по ее имени, или же к имени необходимо добавлять префикс.

Начнем с тривиального случая:

DECLARE

first\_day DATE;

last\_day DATE;

BEGIN

first\_day := SYSDATE;

last\_day := ADD\_MONTHS (first\_day, 6);

END;

Обе переменные first\_day и last\_day объявляются в том же блоке, где они используются, поэтому при обращении к ним указаны только имена без уточняющих префиксов. Такие идентификаторы называются видимыми. В общем случае видимым идентификатором может быть:

1) идентификатор, объявленный в текущем блоке;

2) идентификатор, объявленный в блоке, который включает текущий блок;

3) отдельный объект базы данных (таблица, представление и т. д.) или объект PL/SQL (процедура, функция), владельцем которого вы являетесь;

4) отдельный объект базы данных или объект PL/SQL, на который у вас имеются соответствующие привилегии и который определяется видимым синонимом;

5) индексная переменная цикла (видима и доступна только внутри цикла).

Уточненные идентификаторы

Типичным примером идентификаторов, невидимых в области кода, где они используются, являются идентификаторы, объявленные в спецификации пакета (имена переменных, типы данных, имена процедур и функций). Чтобы обратиться к такому объекту, необходимо указать перед его именем префикс и точку (аналогичным образом имя столбца уточняется именем таблицы, в которой он содержится). Например:

1 price\_util.compute\_means — программа с именем compute\_means из пакета price\_util.

2 math.pi — константа с именем pi, объявленная и инициализированная в пакете math.

Дополнительное уточнение может определять владельца объекта. Например, выражение

scott.price\_util.compute\_means

обозначает процедуру compute\_means пакета price\_util, принадлежащего пользователю Oracle с учетной записью scott.

Уточнение идентификаторов именами модулей

PL/SQL предоставляет несколько способов уточнения идентификаторов для логического разрешения ссылок. Так, использование пакетов позволяет создавать переменные с глобальной областью действия. Допустим, имеется пакет company\_pkg и в спецификации пакета объявлена переменная с именем last\_company\_id:

PACKAGE company\_pkg

IS

last\_company\_id NUMBER;

...

END company\_pkg;

На переменную можно ссылаться за пределами пакета — необходимо лишь указать перед ее именем имя пакета:

IF new\_company\_id = company\_pkg.last\_company\_id THEN

По умолчанию значение, присвоенное переменной пакетного уровня, продолжает действовать на протяжении текущего сеанса базы данных; оно не выходит из области действия вплоть до разрыва подключения.

Идентификатор также можно уточнить именем модуля, в котором он определен:

PROCEDURE calc\_totals

IS

salary NUMBER;

BEGIN

...

DECLARE

salary NUMBER;

BEGIN

salary := calc\_totals.salary;

END;

...

END;

В первом объявлении создается переменная salary, областью действия которой является вся процедура. Однако затем во вложенном блоке объявляется другой идентификатор с тем же именем. Поэтому ссылка на переменную salary во внутреннем блоке всегда сначала разрешается по объявлению в этом блоке, где переменная видима безо всяких уточнений. Чтобы во внутреннем блоке обратиться к переменной salary, объявленной на уровне процедуры, необходимо уточнить ее имя именем процедуры (cal\_totals.salary). Этот метод уточнения идентификаторов работает и в других контекстах. Что произойдет при выполнении следующей процедуры (order\_id — первичный ключ таблицы orders):

PROCEDURE remove\_order (order\_id IN NUMBER)

IS

BEGIN

DELETE orders WHERE order\_id = order\_id; -- Катастрофа!

END;

Этот фрагмент удалит из таблицы orders все записи независимо от переданного значения order\_id. Дело в том, что механизм разрешения имен SQL сначала проверяет имена столбцов и только потом переходит к идентификаторам PL/SQL. Условие WHERE (order\_id = order\_id) всегда истинно, поэтому все данные пропадают.

Возможное решение проблемы выглядит так:

PROCEDURE remove\_order (order\_id IN NUMBER)

IS

BEGIN

DELETE orders WHERE order\_id = remove\_order.order\_id;

END;

В этом случае при разборе имя переменной будет интерпретировано правильно. (Решение работает даже при наличии в пакете функции с именем remove\_order.order\_id)

В PL/SQL установлен целый ряд правил разрешения конфликтов имен, а этой проблеме уделяется серьезное внимание. И хотя знать эти правила полезно, лучше использовать уникальные идентификаторы, чтобы избежать подобных конфликтов. Старайтесь писать надежный код! Если же вы не хотите уточнять каждую переменную, чтобы обеспечить ее уникальность, вам придется тщательно проработать схему назначения имен для предотвращения подобных конфликтов.

Вложенные программы

Завершая тему вложения, области действия и видимости, стоит упомянуть о такой полезной возможности PL/SQL, как вложенные программы (nested programs). Вложенная программа представляет собой процедуру или функцию, которая полностью размещается в разделе объявлений внешнего блока. Вложенная программа может обращаться ко всем переменным и параметрам, объявленным ранее во внешнем блоке, как показывает следующий пример:

PROCEDURE calc\_totals (fudge\_factor\_in IN NUMBER)

IS

subtotal NUMBER := 0;

/\* Начало вложенного блока (в данном случае процедуры).

| Обратите внимание: процедура полностью размещается

| в разделе объявлений calc\_totals.

\*/

PROCEDURE compute\_running\_total (increment\_in IN PLS\_INTEGER)

IS

BEGIN

/\* Переменная subtotal (см. выше) видима и находится в области действия \*/

subtotal := subtotal + increment\_in \* fudge\_factor\_in;

END;

/\* Конец вложенного блока \*/

BEGIN

FOR month\_idx IN 1..12

LOOP

compute\_running\_total (month\_idx);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Годовой итог: ' || subtotal);

END;

Вложенные программы упрощают чтение и сопровождение кода, а также позволяют повторно использовать логику, задействованную в нескольких местах блока.

Запускаем SQL Developer, пришло время написать и запустить стандартную программу -  "Hello World!". Итак, код реализующий данную программу следующий:

SET SERVEROUTPUT ON

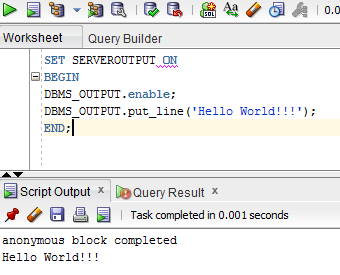
BEGIN

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line('Hello World!!!');

END;

Получаем после компиляции:



Работает. А теперь разберём, что же здесь для чего. Итак строка SET SERVEROUTPUT ON заставляет, сервер выводить сообщения как бы на "экран" или консоль (эту строку нужно вводить один раз на сеанс, при последующих запусках, ее можно не использовать). Хотя, если быть точным, то выводит пакет DBMS\_OUTPUT, а первая строка просто заставляет сервер показывать то, что выводит пакет DBMS\_OUTPUT с помощью метода put\_line(), то есть показывать то, что возвращает сервер.

BEGIN END - это обрамляющий программный блок. Пункт DECLARE отсутствует, в следствии того что нам пока ничего не нужно объявлять. Но в дальнейшем он нам понадобиться. "/" - символ запускающий код на исполнение (применимо к SQL \*Plus, в SQL Developer вы вручную запускаете код). Строка “anonymous block completed" (PL/SQL procedure successfully completed" в SQL \*Plus) просто говорит, что блок выполнился успешно. Вот собственно и все, первая программа написана.

Итак, ваш первый "курсор" в области SGA, совершив работу, выдал "Hello World!". мы просто вывели текст на экран. Но от таких программ мало толку и поэтому сделаем что-то полезное. Как правило, самой простой программой, которой обычно начинают изучать какой либо язык, является калькулятор. Конечно же писать калькулятор на языке PL/SQL - это излишество, но для начала вполне приемлемо. Итак, давайте разберем пару блоков PL/SQL:

DECLARE

A INTEGER;

B INTEGER;

BEGIN

A := 3;

B := 5;

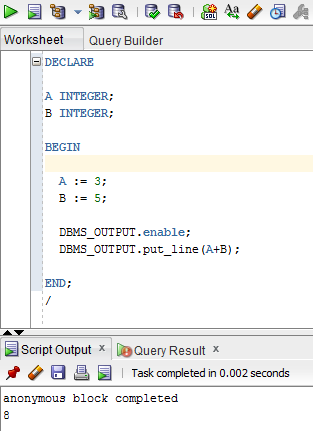
DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(A+B);

END;

/

Получаем следующее:



В разделе DECLARE мы обьявили две переменных - A и B. Все переменные объявляются между оператором DECLARE и BEGIN. Затем в теле блока мы присвоили значениям переменных числа 3 и 5. Инициализация переменных может производиться либо при объявлении, либо в теле блока. Например, можно было сделать вот так:

DECLARE

A INTEGER := 3;

B INTEGER := 5;

BEGIN

.

.

Далее при выводе результата мы прямо в теле метода put\_line объявили сумму двух переменных. Но вернее было бы провести явное преобразование типа с помощью функции TO\_CHAR() вот так:

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(TO\_CHAR(A+B));

В первом примере мы не сделали ошибки, а произошло неявное преобразование типа. Но мой совет делать именно так. В данном случае можно записать более сложное выражение для наглядности и я думаю вам станет яснее почему:

DECLARE

A INTEGER := 3;

B INTEGER;

K VARCHAR2(2) := '12';

BEGIN

B := 5;

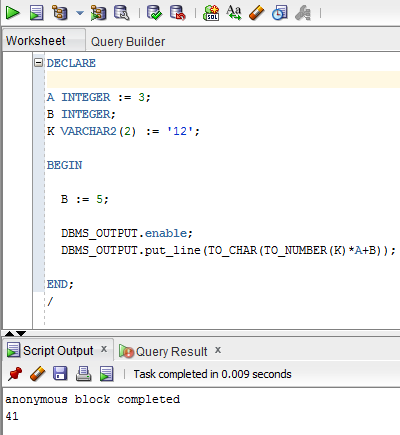
DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(TO\_CHAR(TO\_NUMBER(K)\*A+B));

END;

/

Получаем следующее:



Хорошо виден приоритет унарных операций 12\*3+5 = 41 - верно. Функция TO\_NUMBER(), преобразует строковый литерал K в числовой литерал. Так же мы разнообразно инициализировал переменные, что тоже не плохо при обучении, но лучше делать это одним способом, код будет более читаем. К стати можно записать и так:

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(TO\_CHAR(K\*A+B));

Но все же явное преобразование более понятно. Давайте более ближе посмотрим на литералы. Например можно записать:

DECLARE

A VARCHAR2(1) := ' ';

B VARCHAR2(128) := 'Hello World!!!';

C VARCHAR2(128) := 'How ''are'' you?';

D VARCHAR2(128) := 'Hello Bob - "ok"!';

E VARCHAR2(128) := '12345';

F VARCHAR2(128) := '01/01/1989';

G VARCHAR2(128) := '!@#$%^&\*()\_":;<,.?';

H VARCHAR2(128) := ''' ''';

I NUMBER := 12345;

J NUMBER := -12345;

K NUMBER := 12345.023745;

L NUMBER := 100.;

M NUMBER := 1.0237E2;

N NUMBER := 1.0237E-2;

O NUMBER := 0.34223;

P NUMBER := .321434;

BEGIN

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(A);

DBMS\_OUTPUT.put\_line(B);

DBMS\_OUTPUT.put\_line(C);

DBMS\_OUTPUT.put\_line(D);

DBMS\_OUTPUT.put\_line(E);

DBMS\_OUTPUT.put\_line(F);

DBMS\_OUTPUT.put\_line(G);

DBMS\_OUTPUT.put\_line(H);

DBMS\_OUTPUT.put\_line(TO\_CHAR(I));

DBMS\_OUTPUT.put\_line(TO\_CHAR(J));

DBMS\_OUTPUT.put\_line(TO\_CHAR(K));

DBMS\_OUTPUT.put\_line(TO\_CHAR(L));

DBMS\_OUTPUT.put\_line(TO\_CHAR(M));

DBMS\_OUTPUT.put\_line(TO\_CHAR(N));

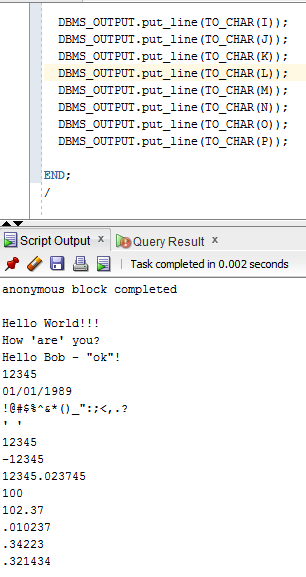
DBMS\_OUTPUT.put\_line(TO\_CHAR(O));

DBMS\_OUTPUT.put\_line(TO\_CHAR(P));

END;

/

Получаем следующее:



Строковые литералы могут быть типов CHAR, VARCHAR2. В строковых литералах компилятор различает регистры символов. Строковые литералы:

A VARCHAR2(1) := ' '; -- пробел

B VARCHAR2(128) := 'Hello World!!!'; -- строка

C VARCHAR2(128) := 'How ''are'' you?'; -- строка с выводом символа '

D VARCHAR2(128) := 'Hello Bob - "ok"!'; -- строка с выводом символа "

E VARCHAR2(128) := '12345'; -- это не число, это строка

F VARCHAR2(128) := '01/01/1989'; -- это не дата, это строка. хотя похоже

G VARCHAR2(128) := '!@#$%^&\*()\_":;<,.?'; -- литералы могут содержать и специальные символы

H VARCHAR2(128) := ''' '''; -- строка содержит три символа "' '"

-- (одинарная кавычка, пробел одинарная кавычка)

Числовые литералы:

I NUMBER := 12345; -- целое число

J NUMBER := -12345; -- отрицательное число

K NUMBER := 12345.023745; -- плавающая точка

L NUMBER := 100.; -- с нулевой точностью

M NUMBER := 1.0237E2; -- научная запись

N NUMBER := 1.0237E-2; -- отрицательная степень

O NUMBER := 0.34223; -- ведущий ноль не нужен

P NUMBER := .321434; -- можно и так

Хорошим стилем программирования является объявление и инициализация переменных. Если инициализировать переменную нет необходимости, то инициализацию можно опустить.

Продолжаем разбираться с переменными. Итак, еще раз внимательнее посмотрим, как их правильно записывают и, как и где инициализируют. Рассмотрим такую последовательность:

DECLARE -- Открывается неименованный блок.

Price NUMBER(5,2) := 12.43; -- Переменная типа NUMBER (с плавающей точкой

-- с точностью 5ть знаков, два после запятой.

Sytki NUMBER := 123; -- Переменная типа NUMBER простая.

LGos INTEGER := 2; -- Переменная типа INTEGER подкласс NUMBER типа.

Max\_Dist CONSTANT REAL := 0.45; -- Константа типа REAL.

-- Тип CONSTANT требует обязательной инициализации!!!

Dis\_Tp VARCHAR2(1) := NULL; -- Строковая переменная типа VARCHAR2 (присвоено значение NULL)

Ds\_Nm CHAR(20) := NULL -- Строковая переменная типа CHAR (присвоено значение NULL)

Tk\_Ir VARCHAR2(50) NOT NULL := 'Hello World!!!'; -- Строковая переменная

-- типа VARCHAR2 не может иметь значение NULL!!!

Tk\_Sr BOOLEAN NOT NULL := TRUE; -- БУЛЕВА! переменная типа BOOLEAN не может иметь значение NULL!!!

It\_Nm VARCHAR2(50) DEFAULT 'Hummer'; -- Строковая переменная типа VARCHAR2 имеет значение по умолчанию

Теперь напишем неименованный блок и посмотрим как это работает:

SET SERVEROUTPUT ON

DECLARE

Price NUMBER(5,2) := 12.43;

Sytki NUMBER := 123;

LGos INTEGER := 2;

Max\_Dist CONSTANT REAL := 0.45;

Dis\_Tp VARCHAR2(1) := NULL;

Ds\_Nm CHAR(20) := NULL;

Tk\_Ir VARCHAR2(50) NOT NULL := 'Hello World!!!';

Tk\_Sr BOOLEAN NOT NULL := TRUE;

It\_Nm VARCHAR2(50) DEFAULT 'Hummer';

BEGIN

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(TO\_CHAR(Price));

DBMS\_OUTPUT.put\_line(TO\_CHAR(Sytki));

DBMS\_OUTPUT.put\_line(TO\_CHAR(LGos));

DBMS\_OUTPUT.put\_line(TO\_CHAR(Max\_Dist));

DBMS\_OUTPUT.put\_line(Dis\_Tp);

DBMS\_OUTPUT.put\_line(Ds\_Nm);

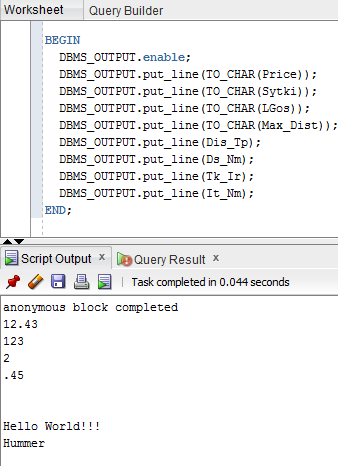
DBMS\_OUTPUT.put\_line(Tk\_Ir);

DBMS\_OUTPUT.put\_line(It\_Nm);

END;

/

Получаем такой вывод



Первые три переменных просто вывелись на экран. Это числовые переменные разных типов и точности. А вот четвертую переменную стоит рассмотреть по ближе. Это CONSTANT переменная, то есть константа. Кроме того, она имеет тип REAL, являющимся разновидностью NUMBER. При таком типе объявления переменная должна сразу же получать значение, то есть инициализироваться. Запомните, все константы должны инициализироваться при объявлении.

Пятая и шестая переменные - это просто строковые литералы разных типов. Седьмая переменная - это строковый литерал, которые не может иметь значение NULL. Так обычно определяют первичный ключ в таблице. Здесь тоже самое просто применяется, при объявлении переменной.

А, вот восьмая переменная - это BOOLEAN переменная, с грифом NOT NULL. Вообще в PL/SQL тип BOOLEAN привнесен, немного искусственно для того, чтобы он был полноценным языком программирования. Как вам, наверное, известно, в Oracle, как и во многих SQL серверах в таблицах нельзя объявить тип поля (столбца) BOOLEAN. В этом смысле обычно применятся поле VARCHAR2(1) и просто записываю туда либо T, либо F, либо NUMBER(1) и значения 0 и 1. А не иметь в самом PL/SQL типа BOOLEAN, было бы не очень естественно. Поэтому он есть. TRUE соответствует 1, а FALSE соответствует 0, как в языке C++, чего и следовало ожидать.

Девятая переменная, это тоже строковый литерал, но он кроме этого имеет значение по умолчанию, такое же объявление можно использовать, когда создается таблица БД. Вот так, если кратко объявляются и инициализируются переменные в PL/SQL. Практически все способы, которые вам могут понадобиться в дальнейшем мы рассмотрели.

Все правила работы с BOOLEAN переменными и константами, те же что и для составных условий WHERE. Есть еще функция XOR она выглядит, для троичной логики так:

X Y X XOR Y

------------------------

TRUE TRUE FALSE

TRUE FALSE TRUE

TRUE NULL NULL

FALSE TRUE TRUE

FALSE FALSE FALSE

FALSE NULL NULL

NULL TRUE NULL

NULL FALSE NULL

NULL NULL NULL

**Операторы ветвления**

Разберём как работают операторы ветвления. Самым простым в использовании, в PL/SQL является оператор IF. Его основная логическая форма имеет вид:

IF(некоторое условие справедливо) THEN

-- условие справедливо, выполнять это.

ELSE -- условие не выполняется

-- выполнять оператор в этой части.

END IF; -- конец условного оператора.

Первый блок оператора после IF-THEN называется "ПРЕДЫДУЩИМ", а блок следующий за ELSE "ПОСЛЕДУЮЩИМ". Каждый оператор IF-THEN может содержать обрамляющие блоки BEGIN - END, если в этом есть необходимость, так как при записи типа:

IF(некоторое условие справедливо) THEN

-- оператор 1

-- оператор 2

-- оператор 3

-- оператор 4

END IF; -- конец условного оператора.

Можно уверенно сказать, что все четыре оператора выполнятся наверняка. Но, если применить такую конструкцию:

IF(некоторое условие справедливо) THEN

BEGIN

-- оператор 1

-- оператор 2

-- оператор 3

-- оператор 4

END;

END IF; -- конец условного оператора.

Код будет более нагляден, хотя это вопрос относится к стилю программирования. Так же считается, хорошим стилем наличие комментария при использовании оператора IF. Особенно в случае большого количества вложений. Оператор IF можно вкладывать на любую глубину выражения. Таким образом, можно задать достаточно сложную логику выражения. Так же для экономии памяти, возможно использовать конструкцию, типа:

IF(некоторое условие справедливо) THEN -- проверка условия

BEGIN

-- условие справедливо, выполнять это.

.

.

END;

ELSE -- условие не выполняется

DECLARE

x NUMBER;

BEGIN

.

.

END;

END IF; -- конец условного оператора.

В этом случае переменная x будет создана, если условие предыдущего блока ложно. Хотя использовать, такие конструкции не всегда оправдано. Так же условные операторы в PL/SQL вычисляются с помощью так называемой "сокращенной" оценки. Допустим, есть такое условие:

IF( a AND b ) THEN -- проверка условия

BEGIN

.

.

END;

END IF; -- конец условного оператора.

Если выражение a равно FALSE, то дальнейшее выражение не вычисляется. Все выражения вычисляются слева направо, а выражения в скобках, имеют наивысший приоритет. Например, вот так:

IF( (a OR f) AND (c OR k) ) THEN -- проверка условия

BEGIN

.

.

END;

END IF; -- конец условного оператора.

Выражения в скобках вычисляются первыми.

**Работа с NULL, ELSIF**

Оператор IF работает с NULL также как выражениях. Если записать:

IF(X = NULL) THEN -- проверка условия

.

.

END IF; -- конец условного оператора.

IF(X != NULL) THEN -- проверка условия

.

.

END IF; -- конец условного оператора.

то условие будет иметь значение FALSE и ничего выполняться не будет. Правильная запись будет такая:

IF(X IS NULL) THEN -- проверка условия

.

.

END IF; -- конец условного оператора.

IF(X IS NOT NULL) THEN -- проверка условия

.

.

END IF; -- конец условного оператора.

NULL также является оператором, который обозначает ничего не делать. Нельзя написать такой блок

DECLARE

begin

end;

но можно

DECLARE

begin

null;

end;

Также можно использовать для явного указания блока ELSE в операторе ветвления

if true then

dosomeaction();

else

null;

end if;

Таким образом явно даётся понять другим разработчикам что именно ничего не надо делать в ином случае, а не просто вы забыли обработать блок ELSE.

А, что делать, если необходимо проверить одно значение несколько раз? Например, вот так:

IF( val = 1 ) THEN -- проверка условия

.

.

END IF; -- конец условного оператора.

.

.

.

IF( val = 9 ) THEN -- проверка условия

.

.

END IF; -- конец условного оператора.

Такое количество операторов IF конечно можно использовать, но это слишком громоздко и не верно, так как существует конструкция IF - THEN - ELSIF. работает она достаточно просто и ее для таких целей вполне достаточно:

IF ( val = 1 ) THEN -- проверка условия

.

.

ELSIF ( val = 2 ) THEN

.

.

ELSIF ( val = 3 ) THEN

.

.

ELSIF ( val = 9 ) THEN

.

.

[ELSE -- не сработало не одно из условий!

]

END IF; -- конец условного оператора.

Очень похоже на оператор CASE в языке Pascal. Но не совсем, хотя выполняет ту же функцию. Секция ELSE срабатывает, если не выполнилось ни одно из условий. Секция ELSE необязательна.

Также циклы можно вкладывать друг в друга

IF условие1

THEN

IF условие2

THEN

команды2

ELSE

IF условие3

THEN

команды3

ELSIF условие4

THEN

команды4

END IF;

END IF;

END IF;

Простые команды CASE

Простая команда CASE позволяет выбрать для выполнения одну из нескольких последовательностей команд PL/SQL в зависимости от результата вычисления выражения. Он записывается следующим образом:

CASE выражение

WHEN результат\_1 THEN

команды\_1

WHEN результат\_2 THEN

команды\_2

...

ELSE

команды\_else

END CASE;

Ветвь ELSE здесь не обязательна. При выполнении такой команды PL/SQL сначала вычисляет выражение, после чего результат сравнивается с результат\_1. Если они совпадают, то выполняются команды\_1. В противном случае проверяется значение результат\_2 и т. д.

Приведем пример простой команды CASE, в котором премия начисляется в зависимости от значения переменной employee\_type:

CASE employee\_type

WHEN 'S' THEN

award\_salary\_bonus(employee\_id);

WHEN 'H' THEN

award\_hourly\_bonus(employee\_id);

WHEN 'C' THEN

award\_commissioned\_bonus(employee\_id);

ELSE

RAISE invalid\_employee\_type;

END CASE;

В этом примере присутствует явно заданная секция ELSE, однако в общем случае она не обязательна. Без секции ELSE компилятор PL/SQL неявно подставляет такой код:

ELSE RAISE CASE\_NOT\_FOUND;

Иначе говоря, если не задать ключевое слово ELSE и если никакой из результатов в секциях WHEN не соответствует результату выражения в команде CASE, PL/SQL инициирует исключение CASE\_NOT\_FOUND. В этом и заключается отличие данной команды от IF. Когда в команде IF отсутствует ключевое слово ELSE, то при невыполнении условия не происходит ничего, тогда как в команде CASE аналогичная ситуация приводит к ошибке.

Интересно посмотреть, как с помощью простой команды CASE реализовать описанную в начале главы логику начисления премий. На первый взгляд это кажется невозможным, но подойдя к делу творчески, мы приходим к следующему решению:

CASE TRUE

WHEN salary >= 10000 AND salary <=20000 THEN

give\_bonus(employee\_id, 1500);

WHEN salary > 20000 AND salary <= 40000 THEN

give\_bonus(employee\_id, 1000);

WHEN salary > 40000 THEN

give\_bonus(employee\_id, 500);

ELSE

give\_bonus(employee\_id, 0);

END CASE;

Здесь важно то, что элементы выражение и результат могут быть либо скалярными значениями, либо выражениями, результатами которых являются скалярные значения. Вернувшись к команде IF...THEN...ELSIF, реализующей ту же логику, вы увидите, что в команде CASE определена секция ELSE, тогда как в команде IF–THEN–ELSIF ключевое слово ELSE отсутствует. Причина добавления ELSE проста: если ни одно из условий начисления премии не выполняется, команда IF ничего не делает, и премия получается нулевой. Команда CASE в этом случае выдает ошибку, поэтому ситуацию с нулевым размером премии приходится программировать явно. Чтобы предотвратить ошибки CASE\_NOT\_FOUND, убедитесь в том, что при любом значении проверяемого выражения будет выполнено хотя бы одно из условий.

Поисковая команда CASE

Поисковая команда CASE проверяет список логических выражений; обнаружив выражение, равное TRUE, выполняет последовательность связанных с ним команд. В сущности, поисковая команда CASE является аналогом команды CASE TRUE, пример которой приведен в предыдущем разделе. Поисковая команда CASE имеет следующую форму записи:

CASE

WHEN выражение\_1 THEN

команды\_1

WHEN выражение\_2 THEN

команда\_2

...

ELSE

команды\_else

END CASE;

Она идеально подходит для реализации логики начисления премии:

CASE

WHEN salary >= 10000 AND salary <=20000 THEN

give\_bonus(employee\_id, 1500);

WHEN salary > 20000 AND salary <= 40000 THEN

give\_bonus(employee\_id, 1000);

WHEN salary > 40000 THEN

give\_bonus(employee\_id, 500);

ELSE

give\_bonus(employee\_id, 0);

END CASE;

Как видите, на первый взгляд, все достаточно просто, хотя и реализует всю функциональность самого языка. Данный оператор является первым фундаментальным оператором языка PL/SQL. Что ж теперь, давайте изобразим небольшую программу для закрепления. Набираем следующее:

-- OPER VARCHAR2(5) := '&TODO';

SET SERVEROUTPUT ON

DECLARE

A INTEGER := 7;

B INTEGER := 4;

OPER VARCHAR2(2) := '+';

BEGIN

DBMS\_OUTPUT.enable;

IF (OPER = '+') THEN

DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'sum = '||TO\_CHAR(A+B));

ELSIF (OPER = '-') THEN

DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'res = '||TO\_CHAR(A-B));

ELSIF (OPER = '\*') THEN

DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'mul = '||TO\_CHAR(A\*B));

ELSIF (OPER = '/') THEN

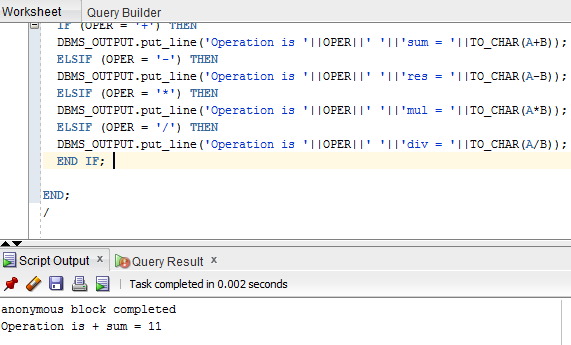
DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'div = '||TO\_CHAR(A/B));

END IF;

END;

/

После запуска получаем:



В данном случае мы применили конструкцию IF - THEN - ELSIF для создания чего-то похожего на калькулятор. В операторе DBMS\_OUTPUT.put\_line - для сцепления строк применяем конструкцию конкатенации строк вида "||" в результате вот такого выражения ('Operation is '||OPER||' '||'sum = '||TO\_CHAR(A+B)) мы получаем строку "Operation is + sum = 11". Таким образом, происходит сцепление отдельных строковых литералов. В проверках условий в операторах IF - THEN - ELSIF хорошо видно, что производится сравнение переменной со строковым литералом, такие конструкции так же применимы при осуществлении проверок. Давайте немного усложним код программы, заменив выражение OPER VARCHAR2(2) := '+'; на OPER VARCHAR2(5) := '&TODO'; вот так:

SET SERVEROUTPUT ON

DECLARE

A INTEGER := 7;

B INTEGER := 4;

OPER VARCHAR2(5) := '&TODO';

BEGIN

DBMS\_OUTPUT.enable;

IF (OPER = '+') THEN

DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'sum = '||TO\_CHAR(A+B));

ELSIF (OPER = '-') THEN

DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'res = '||TO\_CHAR(A-B));

ELSIF (OPER = '\*') THEN

DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'mul = '||TO\_CHAR(A\*B));

ELSIF (OPER = '/') THEN

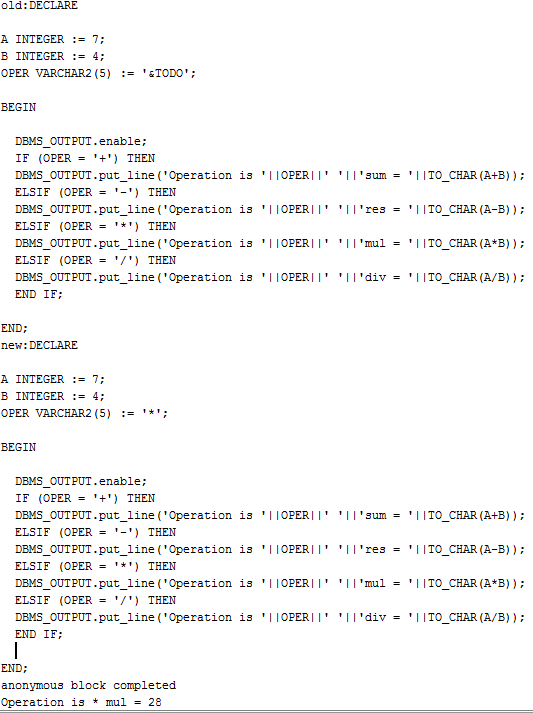
DBMS\_OUTPUT.put\_line('Operation is '||OPER||' '||'div = '||TO\_CHAR(A/B));

END IF;

END;

/

После запуска



Вот теперь наш калькулятор стал задавать нам вопросы, а виноват в этом литерал &, он трактуется как ждать ввода связанной переменной с именем "todo". В данном случае "todo", есть так называемая "связанная" переменная. Домашнее задание переписать код для ввода, сначала чисел, а затем операции над ними.

**GOTO**

Теперь познакомимся с оператором PL/SQL - GOTO. В обычных условиях, применение этого оператора, как и во многих языках программирования не приветствуется. В следствии того, что код вашей программы теряет свою блочную структуру и такой код трудно сопровождать. Тем не менее, знать как работает этот оператор вам пригодится. Давайте посмотрим на следующий фрагмент кода:

BEGIN

BEGIN

GOTO MID;

END;

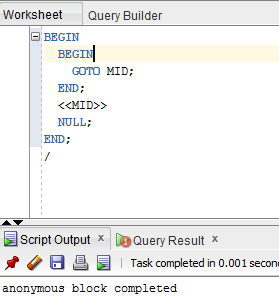
<<MID>>

NULL;

END;

/

Собственно **<<MID>>** есть метка для указания места перехода оператора **GOTO**. Так она объявляется. Двойные кавычки после метки не ставятся. Оператор **NULL** является заполнителем, вместо него вы могли бы вставить свой код. Таким образом данный код легко выполняется, но ничего не производит:



Рассмотрим несколько другой случай:

BEGIN

GOTO MID;

BEGIN

<<MID>>

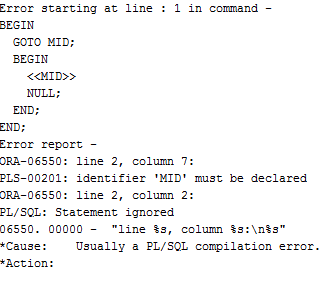
NULL;

END;

END;

/

В результате получаем:



Здесь на лицо нарушение области видимости для оператора GOTO. Внутри блока переменная <<MID>> для него не доступна, что и приводит к ошибке компиляции. То есть внешний блок не имеет информации о содержимом вложенного блока. Так же нельзя переходить из одного блока в другой:

BEGIN

BEGIN

GOTO OTHER;

END;

BEGIN

<<OTHER>>

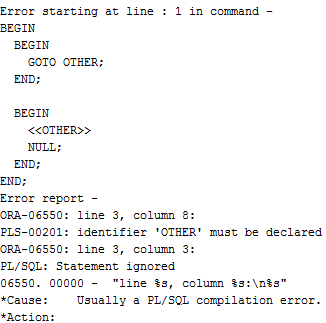
NULL;

END;

END;

/

Получаем:



Что собственно и требовалось доказать. Нельзя производить переход из предшествующих блоков, условных операторов в последующий блок. Вот так:

BEGIN

IF (TRUE) THEN

GOTO OTHER;

ELSE

<<OTHER>>

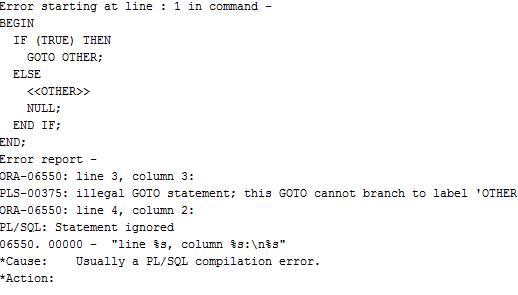
NULL;

END IF;

END;

/

Получаем:



Видно, что блок выполнен с ошибкой компиляции.

Так же остерегайтесь вот таких конструкций:

BEGIN

<<INFINITE\_LOOP>>

GOTO INFINITE\_LOOP

END;

/

PL/SQL не распознает бесконечные циклы и поэтому не пытайтесь выполнить данный код. Иначе придется вручную удалять сеанс PL/SQL. Еще один не маловажный аспект, для того, чтобы писать читаемый код, как правило необходимо использовать комментарии. В PL/SQL есть два способа определения комментариев в коде. Первое это:

-- Это строка комментария!

И второй это как в языке **ANSI C**:

/\* Это комментарий \*/

Как правило, такого типа определений вполне достаточно. Остальное остается на ваше усмотрение. Теперь давайте рассмотрим такой вопрос, ранее мы определяли в наших пробных программах переменные различных типов. Рассмотрим основные из них

* **CHAR**
* **VARCHAR2**
* **NUMBER**
* **DATE**

А вот их основные определения:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тип** | **Подтип** | **Описание** | **Ограничение значений** | **Ограничения для БД** |
| CHAR | CHARACTER STRING ROWID NCHAR | Символьные строки фиксированной длины | Возможный размер 0-32767 байт; по умолчанию 1 | 255 байт |
| VARCHAR2 | VARCHAR STRING+ NVARCHAR | Символьные строки переменной длины | Возможный размер 0-32767 байт; по умолчанию 1 | 4000 байт |
| NUMBER(p,s) | NUMERIC DEC DECIMAL INT INTEGER FLOAT REAL DOUBLE PRECISION SMALLINT | Упакованные десятичные значения. p - общее число цифр. s - масштаб. | Диапазон: 1.0E-129 - 9.99E125 Точность: 1-38 (по умолчанию максимальное значение поддерживаемое системой) Масштаб: -84-127 (по умолчанию 0) | То же что и для PL/SQL |
| DATE |  | Внутреннее представление даты. | 1 января 4712 года до н.э. время в секундах начиная с полуночи (по умолчанию 0:00) | То же что и для PL/SQL |

**Циклы**

Как и во всех языках программирования, в PL/SQL имеются операторы циклов. Их три основных типа:

1. Безусловные циклы (выполняемые бесконечно)
2. Интерактивные циклы (FOR)
3. Условные циклы (WHILE)

Самый простой тип цикла в языке PL/SQL выглядит следующим образом:

LOOP

NULL;

END LOOP

/

Но использовать такой цикл нет смысла, да и для экземпляра БД это не безопасно. Для этого необходимо использовать определенные пути выхода из циклов. Их тоже три:

1. EXIT - Безусловный выход из цикла. Используется посредством применения оператора IF.
2. EXIT WHEN - Выход при выполнении условия.
3. GOTO - Выход из цикла во внешний контекст.

Давайте рассмотрим пример с применением цикла LOOP EXIT WHEN. Запишем следующее:

DECLARE

i NUMBER := 0;

BEGIN

LOOP -- start loop 1

i := i + 1;

IF (i >= 100) THEN

i := 0;

EXIT; -- exit when i >= 0

END IF;

END LOOP; -- end loop 1

LOOP -- start loop 2

i := i + 1;

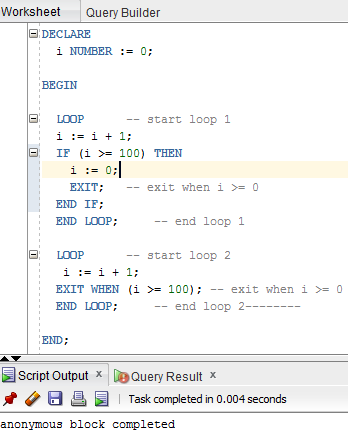
EXIT WHEN (i >= 100); -- exit when i >= 0

END LOOP; -- end loop 2--------

END;

/

Получаем после исполнения:



Видимых действий не было, но и ошибок то же. Первый цикл закончился после того как i стало равно 100. При этом оно получило значение 0 и произошел выход из цикла. Второй цикл применил вложенное предложение EXIT WHEN, что является более верным его использованием синтаксически. Тем не менее, применение условных операторов предполагает перед выходом из цикла проделать некоторые действия. Цикл LOOP EXIT WHEN END LOOP в последующем будет самым, часто используемым при построении конструкций курсоров. Рассмотрим еще одну разновидность вышеприведенного цикла:

DECLARE

k NUMBER := 0;

BEGIN

WHILE (k < 10) LOOP

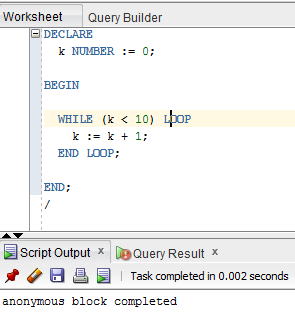
k := k + 1;

END LOOP;

END;

/

Получаем:



Здесь в отличие от предыдущего цикла, действия выполняются до тех пор пока условие истинно. Если условие ложно, то цикл прекращается.

В PL/SQL в конструкциях циклов нет такого, иногда полезного, оператора как CONTINUE, вследствие того, что выражение CONTINUE зарезервировано языком PL/SQL и используется для других целей. Но такую конструкцию как CONTINUE можно эмулировать, применив цикл вида LOOP EXIT WHEN END LOOP и используя весьма не популярный, но в данном случае очень полезный оператор GOTO.

Запишем следующее, выведем все нечетные числа до 20:

SET SERVEROUTPUT ON

DECLARE

s NUMBER := 0;

BEGIN

DBMS\_OUTPUT.enable;

LOOP

IF(MOD(s, 2) = 1) THEN

GOTO LESS;

END IF;

DBMS\_OUTPUT.put\_line(TO\_CHAR(s)||' is even!');

<<LESS>>

EXIT WHEN (s = 20);

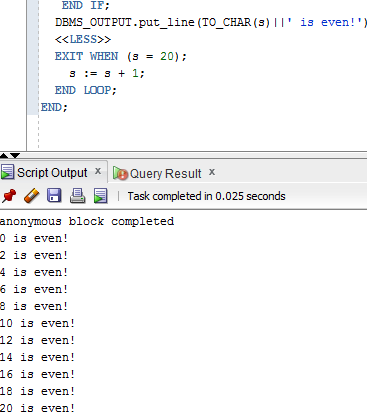
s := s + 1;

END LOOP;

END;

/

Получаем



В результате применения GOTO в теле цикла получаем не сложную и понятную логику работы.

Теперь давайте рассмотрим, не менее полезный и популярный в PL/SQL цикл FOR. Он очень удобен при работе с курсорами, но об этом чуть позднее. Запишем следующее:

BEGIN

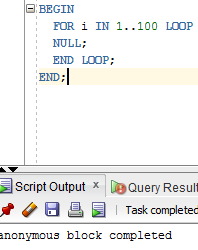
FOR i IN 1..100 LOOP

NULL;

END LOOP;

END;

/



В данный момент FOR успешно ничего не делал сто раз. Итак, давайте рассмотрим его чуть ближе, IN как и в операторе SELECT задает диапазон значений итерации цикла, а ".." это так называемый "оператор диапазона". Замечу так же, что переменная цикла i является переменной только для чтения. Поэтому шаг цикла FOR принудительно изменить нельзя. Если это необходимо, то лучше применять цикл вида LOOP EXIT WHEN END LOOP. Теперь давайте поработаем с числами:

DECLARE

s NUMBER := 0;

BEGIN

DBMS\_OUTPUT.enable;

FOR i IN 1..20 LOOP

IF(MOD(i, 2) = 1) THEN

DBMS\_OUTPUT.put\_line(TO\_CHAR(i)||' is odd!');

s := i;

END IF;

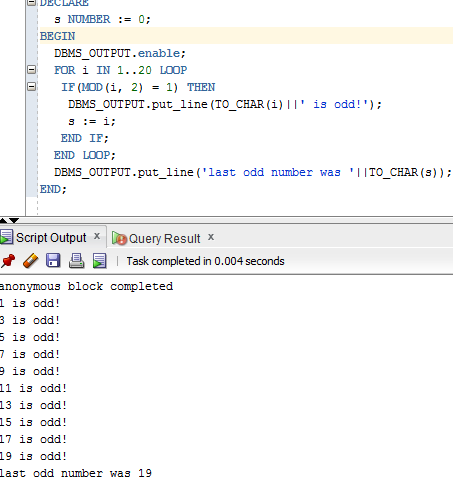
END LOOP;

DBMS\_OUTPUT.put\_line('last odd number was '||TO\_CHAR(s));

END;

/

Получаем:



Та же задачка, только с циклом FOR. Функция MOD возвращает остаток от деления чисел.

Так же в операторе FOR есть возможность задавать обратный отсчет.

BEGIN

DBMS\_OUTPUT.enable;

FOR i IN REVERSE 1..10 LOOP

DBMS\_OUTPUT.put\_line(TO\_CHAR(i)||'-');

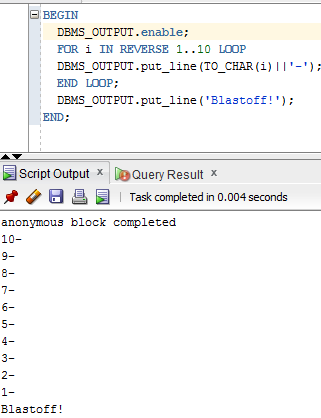
END LOOP;

DBMS\_OUTPUT.put\_line('Blastoff!');

END;

/

Получаем:



**%TYPE и подтипы**

Основная единица хранения информации в БД это таблица. При работе с таблицами в PL/SQL, обычно создается множество переменных, разных типов. В них вы храните какие-либо данные, как правило, из полей таблиц. Например, учебная таблица OFFICES, определенная с помощью оператора CREATE TABLE выглядит следующим образом:

CREATE TABLE OFFICES

(

OFFICE INTEGER PRIMARY KEY,

CITY VARCHAR2(30) NOT NULL,

REGION VARCHAR2(30) NOT NULL,

MGR INTEGER,

TARGET NUMBER,

SALES NUMBER NOT NULL

)

/

Допустим, вы будете хранить содержимое поля REGION, в переменной REGION\_MY и поле MGR в переменной MGR\_MY. Соответственно вы объявите эти переменные примерно следующим образом:

DECLARE

REGION\_MY VARCHAR2(30);

MGR\_MY INTEGER;

.

.

.

И, в какой-то момент, вам нужно переопределить поля таблицы OFFICES, MGR и REGION таким образом:

.

REGION VARCHAR2(128) NOT NULL,

MGR NUMBER(10,5),

.

.

Следовательно, вы уже не сможете в дальнейшем хранить в этих переменных значения переопределенных полей таблицы OFFICES, так как их типы не совпадают и PL/SQL выдаст сообщение об ошибк. А, если таких переменных объявлено достаточно много, то разобраться и изменить все типы будет достаточно сложно и, как правило, вы наделаете кучу ошибок. Для того, чтобы избежать таких неприятностей в дальнейшем необходимо использовать конструкцию %TYPE, следующим образом:

-------- VARIABLE TABLE.FIELD%TYPE --------------

Или, если более понятно, то вышеуказанные переменные нужно объявить вот так:

DECLARE

REGION\_MY OFFICES.REGION%TYPE;

MGR\_MY OFFICES.MGR%TYPE;

BEGIN

.

.

.

Конструкция получается более громоздкая но зато точная и надежная. Теперь, если тип поля изменится, то изменится и тип всех переменных связанных с данным полем. Можно привести еще несколько примеров применения конструкции **%TYPE**:

DECLARE

v\_MYCITY OFFICES.CITY%TYPE;

v\_MYSALES OFFICES.SALES%TYPE;

v\_TEMPOLD NUMBER(7,4) NOT NULL;

v\_DOPTEMP v\_TEMPOLD%TYPE;

Обратите внимане, что для переменной v\_TEMPOLD есть ограничение NOT NULL, а вот для переменной v\_DOPTEMP это ограничение не переносится, так как применяется конструкция %TYPE, от типа другой переменной. Применение конструкции %TYPE, является хорошим стилем программирования, так как делает код более понятным и проще адаптируемым. Вот собственно, что касается, конструкции %TYPE.

В PL/SQL имеется так же возможность объявлять так называемы подтипы (subtype). То есть типы на основе других, стандартных типов. Такое объявление, как правило применяется для того, чтобы получить более понятное описание типа. Ряд типов в PL/SQL, такие например как INTEGER, DECIMAL - являются подтипом, типа NUMBER.

Делается это таким образом:

---- SUBTYPE --- новый тип -- IS -- исходный тип -----------------------

В части исходный тип, можно применить любой стандартный тип или конструкцию %TYPE. Например:

DECLARE

SUBTYPE t\_LoopCn IS NUMBER; -- Определяем новый тип

v\_LpCounter t\_LoopCn; -- Объявляем переменную с эти подтипом

Сразу замечу, что подтип нельзя ограничивать непосредственно в определении SUBTYPE. Это будет ошибкой:

DECLARE

SUBTYPE t\_LoopCn IS NUMBER(4); -- Error !!!

v\_LpCounter t\_LoopCn; -- Объявляем переменную с эти подтипом

Но данное ограничение можно обойти, применив фиктивную переменную нужного типа (с ограничением) и в определении SUBTYPE воспользоваться конструкцией %TYPE:

DECLARE

v\_NullVar NUMBER(7,3);

SUBTYPE t\_LoopCn IS v\_NullVar%TYPE; -- Определяем новый подтип

v\_LpCounter t\_LoopCn; -- Объявляем переменную с этим подтипом

При объявлении переменной с неограниченным подтипом ее тип может быть ограничен, вот так:

DECLARE

SUBTYPE t\_NumVar IS NUMBER -- Определяем неограниченный подтип,

v\_Temp t\_NumVar(5); -- но ограничиваем переменную с этим подтипом!

При этом не забывайте, что подтип принадлежит к тому же семейству типов, что и базовый тип.

**Курсоры**

Основной базис PL/SQL мы с вами, можно сказать, получили. Пришло время начинать разбираться с краеугольным камнем языка PL/SQL - а именно с таким понятием как "КУРСОР".

Итак, в прошлый раз мы изучили оператор %TYPE. Так же в PL/SQL для удобства работы с данными имеется еще один очень полезный и наиболее часто применимый оператор %ROWTYPE. Посмотрим, как он работает:

---------- переменная - структура данных%ROWTYPE --------------

Запишем такой пример:

DECLARE

v\_RecOffices OFFICES%ROWTYPE;

.

.

.

При этом переменная v\_RecOffices будет иметь вот такую внутреннюю структуру:

v\_RecOffices(

OFFICE INTEGER

CITY VARCHAR2(30),

REGION VARCHAR2(30),

MGR INTEGER,

TARGET NUMBER,

SALES NUMBER

)

Так как мы объявили переменную v\_RecOffices на базе таблицы OFFICES, то в результате получаем "запись" с внутренней структурой идентичной полям таблицы OFFICES. То есть в своей сути v\_RecOffices это не просто переменная, а переменная типа "запись" и она содержит в нашем случае как бы шесть переменных, типы которых, совпадают с типами полей таблицы OFFICES.

Отсюда следует, что за один раз в такую, переменную, можно будет поместить целую строку из таблицы OFFICES и, если какое либо поле в таблице будет изменено в процессе сопровождения кода, то менять что-то в коде нет необходимости, так как тип %ROWTYPE, будет сам реагировать на эти изменения. Также есть возможность обращаться к любому полю записи v\_RecOffices. Такое обращение производиться через точечную нотацию (как в языке Pascal):

DECLARE

a INTEGER := v\_RecOffices.OFFICE;

b VARCHAR2(50) := v\_RecOffices.CITY;

c INTEGER := v\_RecOffices.MGR;

То есть, после выборки строки из таблицы очень легко получить отдельные значения полей в промежуточные переменные. За один раз можно выбрать только одну строку в переменную данного типа. Так же как и для %TYPE ограничение NOT NULL для полей записи отсутствует. В дальнейшем вы увидите как %ROWTYPE значительно облегчает последовательную выборку данных при работе с курсорами.

Что же такое курсор. По большому счету понятие курсор в серверах SQL является фундаментальным потому что все базовые операторы DML - это не что иное как курсоры. При выполнении INSERT, UPDATE и т.д. в "глобальной системной области" всегда, открывается курсор. Курсоры имеют ряд атрибутов для облегчения работы с ними. Курсоры могут быть явными и не явными. Все операторы DML - это не что иное, как неявные курсоры. Курсоры могут быть параметризованные или нет.

При обработке SQL - операторов Oracle выделяет область памяти называемую "контекстной областью" (context area). Она содержит информацию необходимую для начала и завершения обработки SQL оператора. В том числе: число строк обрабатываемых оператором, указатель на представление этого оператора после синтаксического анализа (еще это называют парсинг) и активный набор (active set) - например набор строк возвращаемых запросом.

Итак: КУРСОР - это указатель (хотя как такового, понятия "указатель" в PL/SQL нет) на контекстную область памяти, с помощью которого программа на языке PL/SQL может управлять контекстной областью и ее состоянием во время обработки оператора.

Объявление курсора определяет какое выражение языка SQL будет передано программе SQL Statement Executor (системе исполнителю выражения SQL). Курсор может представлять собой любое допустимое предложение языка SQL. Так же, курсор является основным базовым "кирпичиком" для построения блоков PL/SQL. Курсоры обеспечивают циклический механизм оперирования наборами данных в БД. Курсор может возвращать одну или несколько строк данных или вообще ни одной.

Типичная последовательность, при операциях (в данном случае с явными (определенными) курсорами) будет такая:

1. Объявление курсора и структуры данных, в которую, будут помещены найденные строки.
2. Открытие курсора.
3. Последовательная выборка данных.
4. Закрытие курсора.

Полный синтаксис определения явного курсора таков:

------ CURSOR -- имя (передаваемые параметры) --- IS -----

---------------- SELECT список полей FROM таблица выбора

---------------- WHERE условия выбора в курсор -----------

Рассмотрим несколько определений курсоров

DECLARE

-- 1.

-- Выбрать все заказы:

CURSOR get\_orders IS

SELECT \* FROM ORDERS;

-- 2.

-- Выбрать несколько столбцов

-- для определенного номера заказа

CURSOR get\_orders(Pord\_num ORDERS.order\_num%TYPE) IS

SELECT ORDER\_DATE, MFR, AMOUNT FROM ORDERS

WHERE order\_num = Pord\_num;

-- 3.

-- Получить полную запись

-- для определенного номера заказа

CURSOR get\_orders(Pord\_num ORDERS.order\_num%TYPE) IS

SELECT \* FROM ORDERS

WHERE order\_num = Pord\_num

RETURN ORDERS%ROWTYPE;

-- 4.

-- Получение имени сотрудника

-- по его номеру

CURSOR get\_name(empl\_nm SALESREPS.empl\_num%TYPE)

RETURN SALESREPS.name%TYPE IS

SELECT name FROM SALESREPS WHERE empl\_num = empl\_nm;

Надеюсь, вы заметили, что курсоры выполняют отбор вполне осмысленной информации. В явных курсорах ключевым звеном является оператор SELECT . Именно он определяет, что будет возвращать курсор после того как вы его откроете.

Курсор под номером 1. Это простой не параметризованный курсор возвращающий все содержимое таблицы запроса в данном случае ORDERS. Это самый легкий из курсоров.

Курсор 2. Это более сложный параметризованный курсор с одним параметром, но их может быть и больше. Он возвращает как видно из условия WHERE одну строку, так как поле order\_num ключевое. Обратите внимание при передаче параметра использована конструкция %TYPE.

Курсор под номером 3. Чуть более расширенная версия предыдущего, кроме того используется конструкция RETURN - с применением переменной записи на базе таблицы ORDERS. Такая конструкция просто возвратит одну полную запись из таблицы ORDERS при открытии курсора. Такие конструкции курсоров мы рассмотрим после того, как разберемся с процедурами и функциями.

Курсор 4. Это то же самое, что и курсор 3, но он применяет конструкцию RETURN для того, чтобы вернуть значение одного поля таблицы SALESREPS, обратите внимание на местоположение оператора RETURN, хотя такое редко используется на практике. Но, тем не менее, в документации такое есть и полезно посмотреть как это выглядит.

Мы рассмотрели первую часть правил работы с курсором, а именно первый пункт (Объявление курсора и структуры данных, в которую, будут помещены найденные строки.) Определение структуры данных в которую, будут помещены найденные строки мы пока опустим, к ней мы еще вернемся, а пока поговорим о втором пункте (Открытие курсора).

Курсор открывается с помощью оператора OPEN. Его синтаксис следующий:

------ OPEN - имя курсора(передаваемые параметры) ---------

Давайте, определим вот такой курсор, следующим образом:

DECLARE

v\_Office OFFICES.OFFICE%TYPE;

-- определяем курсор

CURSOR get\_offices IS

SELECT \* FROM OFFICES

WHERE OFFICE = v\_Office;

BEGIN

-- присваиваем значение

v\_Office := 11;

OPEN get\_offices; -- открываем курсор

-- меняем значение

v\_Office := 12;

END;

/

Здесь определяется простой не параметризированный курсор, который тем не менее, получает условие через объявленную переменную, скажу отметим что это не очень хороший стиль работы, но пока пусть будет так. Что же происходит когда курсор открывается?

1. Анализируется значение переменных привязки.
2. На основе значений переменных привязки определяется активный набор.
3. Указатель активного набора устанавливается на первую строку.

Таким образом переменные привязки анализируются только во время открытия курсора, а у нас переменная v\_Office перед открытием курсора равна 11. Соответственно результирующий набор отвечает данному условию. Дальнейшее изменение переменной v\_Office с 11 на 12 уже не влияет на активный набор запроса. Такой алгоритм открытия называется согласованностью чтения (read - consistency).

Чтобы получить, новый набор нужно закрыть курсор, а затем снова его открыть, вот так:

.

.

.

OPEN get\_offices; -- открываем курсор

CLOSE get\_offices;

-- меняем значение

v\_Office := 12;

OPEN get\_offices; -- открываем курсор

END;

/

Вот теперь курсор получил данные соответствующие новому значению переменной v\_Office. Можно сделать и так:

.

.

.

OPEN get\_offices; -- открываем курсор

-- меняем значение значение

v\_Office := 12;

OPEN get\_offices; -- открываем курсор

END;

/

Так как повторное открытие курсора приводит к неявному вызову оператора CLOSE, то результат будет такой же, но использовать CLOSE, по моему мнению, более правильно, так как код более читаем и это является хорошим стилем программирования. Кроме того, совершенно допустимо открытие одновременно нескольких курсоров. Параметризованные курсоры открываются точно так же, но лишь с той разницей, что им передается заранее определенный параметр. Перепишем наш предыдущий курсор, как параметризованный, что так же является более правильным стилем программирования, вследствие того, что курсор с условием WHERE в части SELECT должен принимать условие отбора как параметр, а не как "размытую" переменную. Запишем:

DECLARE

-- определяем переменную отбора

v\_Office OFFICES.OFFICE%TYPE;

-- определяем параметризованный курсор

CURSOR get\_offices(v\_Off OFFICES.office%TYPE) IS

SELECT \* FROM OFFICES

WHERE OFFICE = v\_Off;

BEGIN

-- присваиваем значение

v\_Office := 11;

OPEN get\_offices(v\_Office); -- открываем курсор и передаем параметр отбора

-- меняем значение

v\_Office := 12;

END;

/

Вот так будет выглядеть параметризованный курсор. Ну, а в остальном, все вышесказанное справедливо и для него. Мы уже видели что оператор **CLOSE** закрывает курсор. Давайте немного подробнее остановимся на нем. Его синтаксис таков:

------ CLOSE - имя курсора ------------

Хорошим стилем программирования является закрытие курсора, когда работа с ним окончена. Закрытие курсора говорит о том, что программа закончила с ним работу и что все ресурсы, которые занимает данный курсор можно освободить. Если в вашей программе или процедуре один курсор, то если вы его не закроете, то система сама при завершении вызовет оператор CLOSE, но если у вас достаточно большой блок обработки с десятком курсоров, то их не закрытие может существенно снизить производительность вашего блока PL/SQL**.** После закрытия курсора, выбирать из него строки уже нельзя. Если попытаться это сделать, то получите сообщение об ошибке:

ORA-1001 Invalid Cursor

или:

ORA-1002 Fetch out of Sequence

ORA-1001 - так же срабатывает, если вы попытаетесь, закрыть уже закрытый курсор. Так что с определением открытием и закрытием курсора будем считать мы разобрались.

Наконец, пора получить какие-нибудь результаты, т.е. данные. Выборка данных из курсора производится с помощью оператора FETCH. Пожалуй, он является ключевой фигурой в данном случае. Так как именно с его помощью происходит выборка из активного набора, сформированного при открытии курсора. После того как курсор открыт, данные в контекстной области SGA уже готовы, их остается только извлечь. Синтаксис оператора FETCH:

---------- FETCH - имя курсора - INTO - список переменных --------------

и

---------- FETCH - имя курсора - INTO - запись PL/SQL (%ROWTYPE) --------

Переходим к практическим действиям. Запускаем SQL Developer. Запишем вот такой блок, спользуя пройденный материал:

DECLARE

v\_Office OFFICES.OFFICE%TYPE;

v\_City OFFICES.CITY%TYPE;

CURSOR get\_offices IS

SELECT OFFICE, CITY

FROM OFFICES;

BEGIN

OPEN get\_offices;

-- Use operator FETCH to get variables!

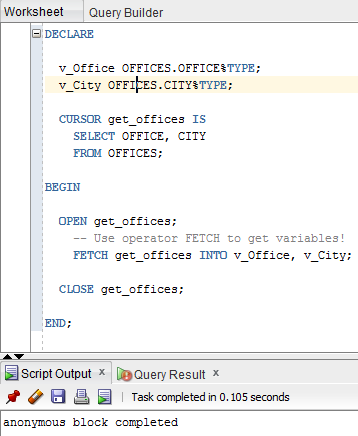
FETCH get\_offices INTO v\_Office, v\_City;

CLOSE get\_offices;

END;

/

Получаем следующее:



Видим что у нас правильная выборка, в две переменные первой строки активного набора. Но, как то не наглядно, не видно результата. Перепишем наш предыдущий блок:

DECLARE

v\_Office OFFICES.OFFICE%TYPE;

v\_City OFFICES.CITY%TYPE;

CURSOR get\_offices IS

SELECT OFFICE, CITY

FROM OFFICES;

BEGIN

OPEN get\_offices;

-- Use operator FETCH to get variables!

FETCH get\_offices INTO v\_Office, v\_City;

DBMS\_OUTPUT.enable;

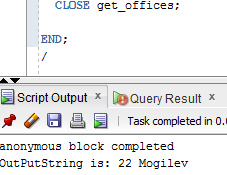
DBMS\_OUTPUT.put\_line('OutPutString is: '||TO\_CHAR(v\_Office)||' '||v\_City);

CLOSE get\_offices;

END;

/

Получаем следующее:



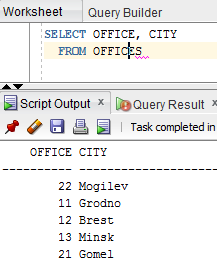
Видим результат. Давайте подробнее рассмотрим, что же произошло. Поcле открытия курсора (не параметризованного) происходит выборка данных, согласно выражения SELECT курсора. Легко проверить, что должен вернуть курсор просто выполнив его оператор SELECT, вот так:

SELECT OFFICE, CITY

FROM OFFICES

/

Получаем:



Это и есть ваш результирующий набор, но здесь пять строк. Да, в таблице OFFICES пять записей, а пример с FETCH вернул одну строку, потому что этот оператор производит выбор одной строки и смещает указатель в контекстной области на единицу. И ждет следующей команды на выборку. До тех пор, пока не будет достигнута последняя запись. Когда будет достигнута последняя запись, сработает атрибут курсора %NOTFOUND (его мы рассмотрим чуть позднее), он станет TRUE. Это значит, что все записи из результирующего набора выбраны. С помощью нашего примера можно выбрать и все пять записей, сделав вот так:

DECLARE

v\_Office OFFICES.OFFICE%TYPE;

v\_City OFFICES.CITY%TYPE;

CURSOR get\_offices IS

SELECT OFFICE, CITY

FROM OFFICES;

BEGIN

OPEN get\_offices;

DBMS\_OUTPUT.enable;

-- Use operator FETCH to get variables!

FETCH get\_offices INTO v\_Office, v\_City; -- 1

DBMS\_OUTPUT.put\_line('OutPutString is: '||TO\_CHAR(v\_Office)||' '||v\_City);

FETCH get\_offices INTO v\_Office, v\_City; -- 2

DBMS\_OUTPUT.put\_line('OutPutString is: '||TO\_CHAR(v\_Office)||' '||v\_City);

FETCH get\_offices INTO v\_Office, v\_City; -- 3

DBMS\_OUTPUT.put\_line('OutPutString is: '||TO\_CHAR(v\_Office)||' '||v\_City);

FETCH get\_offices INTO v\_Office, v\_City; -- 4

DBMS\_OUTPUT.put\_line('OutPutString is: '||TO\_CHAR(v\_Office)||' '||v\_City);

FETCH get\_offices INTO v\_Office, v\_City; -- 5

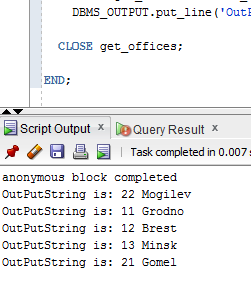
DBMS\_OUTPUT.put\_line('OutPutString is: '||TO\_CHAR(v\_Office)||' '||v\_City);

CLOSE get\_offices;

END;

/

Получаем:



Вот все пять записей, каждый последующий **FETCH** вернул по одной записи. И вы уже, наверное, догадались, что здесь просто напрашивается оператор цикла. Для начала (просто для примера) применим, оператор **LOOP EXIT WHEN** и при этом немного изменим наш курсор, применив атрибут **%ROWTYPE** вот так:

DECLARE

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

v\_gt get\_offices%ROWTYPE;

BEGIN

OPEN get\_offices;

LOOP

EXIT WHEN get\_offices%NOTFOUND;

DBMS\_OUTPUT.enable;

-- Use operator FETCH to get variables!

FETCH get\_offices INTO v\_gt;

DBMS\_OUTPUT.put\_line('Get Data: '||TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

||v\_gt.REGION||' '||TO\_CHAR(v\_gt.MGR)||' '||TO\_CHAR(v\_gt.TARGET)||' '||TO\_CHAR(v\_gt.SALES));

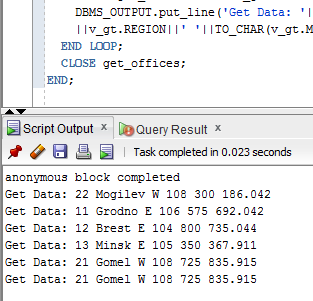
END LOOP;

CLOSE get\_offices;

END;

/

И наконец получаем:



Мы, чуть забежав вперед, с применением оператора цикла и курсорного атрибута %NOTFOUND, написали наш первый полноценный курсор, который выбирает все данные из таблицы и выводит их на экран.

**Аттрибуты курсоров**

Настало время рассмотреть немаловажный аспект, а именно курсорные атрибуты. В PL/SQL имеется четыре основных курсорных атрибута %FOUND, %NOTFOUND, %ISOPEN и %ROWCOUNT. Атрибуты курсора объявляются подобно операторам %TYPE и %ROWTYPE, справа от имени курсора, вот так:

------------ имя курсора%атрибут -------------

Для того, чтобы понять что это за операторы и для чего они предназначены, давайте предположим, что наша учебная таблица OFFICES, содержит два столбца OFFICE и CITY, а так же имеет только две записи. Вот таким образом:

Таблица OFFICES

OFFICE CITY

------- -------------------

22 Mogilev

11 Grodno

Теперь к этой таблице запишем вот такой блок PL/SQL и объявим в нем курсор get\_offices следующим образом:

DECLARE

-- Cursor declaration --

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

-- Record to store the fetched data --

v\_gt get\_offices%ROWTYPE;

BEGIN

-- location (1) -- cursor is open --

OPEN get\_offices;

-- location (2)

FETCH get\_offices INTO v\_gt; -- fetch first row --

-- location (3)

FETCH get\_offices INTO v\_gt; -- fetch second row --

-- location (4)

FETCH get\_offices INTO v\_gt; -- Third first --

-- location (5)

CLOSE get\_offices; -- cursor is close --

-- location (6)

END;

/

Рассмотрим по очереди все атрибуты курсоров. Начнем с атрибута %FOUND. Данный атрибут является логическим объектом и возвращает следующие значения согласно точкам, указанным в нашем курсоре.

%FOUND

|  |  |  |
| --- | --- | --- |
| Точка | Значение get\_offices%FOUND | Пояснение |
| 1. | Ошибка ORA-1001 | Курсор еще не открыт и активного набора для него не существует! |
| 2. | NULL | Хотя курсор открыт, не было произведено ни одного считывания строк. Значение атрибута не определено. |
| 3. | TRUE | С помощью предшествующего оператора FETCH выбрана первая строк таблицы OFFICES. |
| 4. | TRUE | С помощью предшествующего оператора FETCH выбрана вторая строк таблицы OFFICES. |
| 5. | FALSE | Предшествующий оператор FETCH не вернул никаких данных, так как все строки активного набора выбраны. |
| 6. | Ошибка ORA-1001 | Курсор закрыт, и вся хранившаяся информация об активном наборе удалена. |

Атрибут %NOTFOUND является полной противоположностью %FOUND и так же хорошо понятен из приведенной ниже таблицы:

|  |  |  |
| --- | --- | --- |
| Точка | Значение get\_offices%NOTFOUND | **Пояснение** |
| 1. | Ошибка ORA-1001 | Курсор еще не открыт и активного набора для него не существует |
| 2. | NULL | Хотя курсор открыт, не было произведено ни одного считывания строк. Значение атрибута не определено. |
| 3. | FALSE | С помощью предшествующего оператора FETCH выбрана первая строк таблицы OFFICES. |
| 4. | FALSE | С помощью предшествующего оператора FETCH выбрана вторая строк таблицы OFFICES. |
| 5. | TRUE | Предшествующий оператор FETCH не вернул никаких данных, так как все строки активного набора выбраны. |
| 6. | Ошибка ORA-1001 | Курсор закрыт, и вся хранившаяся информация об активном наборе удалена. |

Атрибут %ISOPEN так же логический объект и указывает только на то, открыт ли курсор или нет. Возвращаемые значение приведены ниже

|  |  |  |
| --- | --- | --- |
| Точка | Значение get\_offices%ISOPEN | **Пояснение** |
| 1. | FALSE | Курсор get\_offices еще не открыт. |
| 2. | TRUE | Курсор get\_offices был открыт. |
| 3. | TRUE | Курсор get\_offices еще открыт. |
| 4. | TRUE | Курсор get\_offices еще открыт. |
| 5. | TRUE | Курсор get\_offices еще открыт. |
| 6. | FALSE | Курсор get\_offices закрыт. |

Атрибут %ROWCOUNT является числовым атрибутом и возвращает число строк считанных курсором на определенный момент времени. Его значение приведены ниже:

|  |  |  |
| --- | --- | --- |
| **Точка** | **Значение get\_offices%ROWCOUNT** | **Пояснение** |
| 1. | Ошибка ORA-1001 | Курсор еще не открыт и активного набора для него не существует! |
| 2. | 0 | Хотя курсор открыт, не было произведено ни одного считывания строк. |
| 3. | 1 | Считана первая строка таблицы OFFICES |
| 4. | 2 | Считана вторая строка таблицы OFFICES |
| 5. | 2 | К данному моменту считаны две строки таблицы OFFICES |
| 6. | Ошибка ORA-1001 | Курсор закрыт, и вся хранившаяся информация об активном наборе удалена. |

Теперь стало ясно, как работает последний курсор из прошлого примера, где мы применили атрибут %NOTFOUND в теле цикла, чтобы определить, что пора заканчивать чтение строк. Что ж, применим наши знания на практике. Перепишем, наш учебный курсор вот так:

DECLARE

-- Cursor declaration --

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

-- Record to store the fetched data --

v\_gt get\_offices%ROWTYPE;

BEGIN

DBMS\_OUTPUT.enable;

IF(get\_offices%ISOPEN) THEN

DBMS\_OUTPUT.put\_line('Cursor get\_offices is open now!');

ELSE

DBMS\_OUTPUT.put\_line('Cursor get\_offices is close now!');

END IF;

-- location (1) -- cursor is open --

OPEN get\_offices;

IF(get\_offices%ISOPEN) THEN

DBMS\_OUTPUT.put\_line('Cursor get\_offices is open now!');

ELSE

DBMS\_OUTPUT.put\_line('Cursor get\_offices is close now!');

END IF;

-- location (2)

FETCH get\_offices INTO v\_gt; -- fetch first row --

IF(get\_offices%FOUND) THEN

DBMS\_OUTPUT.put\_line('Row is Found!');

ELSE

DBMS\_OUTPUT.put\_line('Row is not Found!');

END IF;

DBMS\_OUTPUT.put\_line(TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

||'The count row is '||TO\_CHAR(get\_offices%ROWCOUNT));

-- location (3)

FETCH get\_offices INTO v\_gt; -- fetch second row --

IF(get\_offices%NOTFOUND) THEN

DBMS\_OUTPUT.put\_line('Row is not Found!');

ELSE

DBMS\_OUTPUT.put\_line('Row is Found!');

END IF;

DBMS\_OUTPUT.put\_line(TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

||'The count row is '||TO\_CHAR(get\_offices%ROWCOUNT));

-- location (4)

FETCH get\_offices INTO v\_gt; -- Third first --

IF(get\_offices%FOUND) THEN

DBMS\_OUTPUT.put\_line('Row is Found!');

ELSE

DBMS\_OUTPUT.put\_line('Row is not Found!');

END IF;

DBMS\_OUTPUT.put\_line(TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

||'The count row is '||TO\_CHAR(get\_offices%ROWCOUNT));

-- location (5)

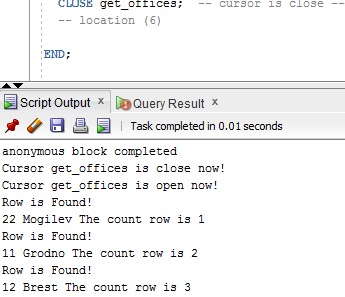
CLOSE get\_offices; -- cursor is close --

-- location (6)

END;

/

Получаем:



Здесь мы применили все атрибуты для того, чтобы на практике показать, как они работают и устроены. На дом задание написать тот же курсор, применив цикл и выбрав все записи из него.

**RECORD**

Далее давайте разберем один довольно интересный тип данных, который очень напоминает структуры в языке C. Обычные скалярные типы, такие как VARCHAR2, NUMBER и т.д. предварительно определены в модуле STANDART. По этому для их использования программе требуется лишь объявить переменную, данного типа, например вот так:

DECLARE

m\_COMPANY VARCHAR2(30);

m\_CUST\_REP INTEGER;

m\_CREDIT\_LIMIT NUMBER;

Но иногда удобнее использовать, так называемый составной тип. Одним из таких типов, в языке PL/SQL является запись RECORD. Посмотрим на синтаксис объявления записи:

-------- TYPE тип записи IS RECORD ( -------------------

-------- поле1 тип1 [NOT NULL] [:= выражение1] -------------

-------- поле2 тип2 [NOT NULL] [:= выражение2] -------------

-------- поле-n тип-n [NOT NULL] [:= выражение-n] ); -------

Как видите для полей записи, можно указывать ограничение NOT NULL, но подобно описанию переменной вне записи исходное значение и ограничение NOT NULL не обязательны. Давайте запишем для примера вот такой блок:

DECLARE

TYPE is\_SmplRec IS RECORD

(

m\_Fld1 VARCHAR2(10),

m\_Fld2 VARCHAR2(30) := 'Buber',

m\_DtFld DATE,

m\_Fld3 INTEGER := 1000,

m\_Fld4 VARCHAR2(100) NOT NULL := 'System'

);

MY\_SMPL is\_SmplRec;

BEGIN

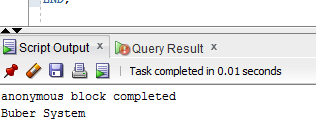
DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(MY\_SMPL.m\_Fld2||' '||MY\_SMPL.m\_Fld4);

END;

/

После его выполнения получаем:



Процедура успешно выполнилась и вернула значения. Здесь хорошо видно, как мы проинициализировали переменные записи внутри объявления. Так же одно из полей объявленное как NOT NULL сразу получило значение, вследствие того, что при этих условиях это необходимо. Рассмотрим блок, где показано, как с использованием правил обращения к полям записи, а именно через точечную нотацию (как в языке Pascal), так же можно переопределить исходные значения полей записи:

DECLARE

TYPE is\_SmplRec IS RECORD

(

m\_Fld1 VARCHAR2(10),

m\_Fld2 VARCHAR2(30) := 'Buber',

m\_DtFld DATE,

m\_Fld3 INTEGER := 1000,

m\_Fld4 VARCHAR2(100) NOT NULL := 'System'

);

MY\_SMPL is\_SmplRec;

BEGIN

MY\_SMPL.m\_DtFld := SYSDATE;

MY\_SMPL.m\_Fld4 := 'Unknown';

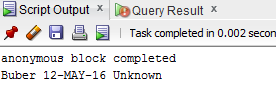
DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(MY\_SMPL.m\_Fld2||' '||MY\_SMPL.m\_DtFld||' '||MY\_SMPL.m\_Fld4);

END;

/

После обработки получаем:



Как видно переменная MY\_SMPL.m\_DtFld получила значение текущей даты, с применением функции SYSDATE, которая возвращает текущую дату и время системы, а переменная MY\_SMPL.m\_Fld4, сменила строковый литерал. Вот таким образом, записи как составной тип данных способны к изменению своего внутреннего содержимого.

Теперь рассмотрим, как производится присваивание записей. В языке PL/SQL при присвоении значения одной записи другой используется семантика копирования. Присвоить одну запись непосредственно другой не допускается, даже если поля в обоих записях одинаковы. Для PL/SQL это разные типы и по этому происходит следующее:

DECLARE

TYPE is\_SmplRecOne IS RECORD

(

m\_Fld1 VARCHAR2(10),

m\_Fld2 VARCHAR2(30),

m\_DtFld DATE,

m\_Fld3 INTEGER,

m\_Fld4 VARCHAR2(100)

);

TYPE is\_SmplRecTwo IS RECORD

(

m\_Fld1 VARCHAR2(10),

m\_Fld2 VARCHAR2(30),

m\_DtFld DATE,

m\_Fld3 INTEGER,

m\_Fld4 VARCHAR2(100)

);

MY\_SMPLONE is\_SmplRecOne;

MY\_SMPLTWO is\_SmplRecTwo;

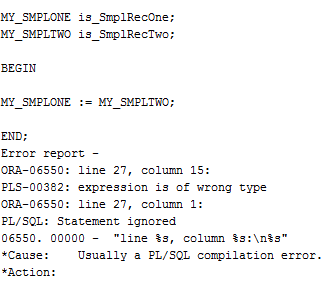
BEGIN

MY\_SMPLONE := MY\_SMPLTWO;

END;

/

Получаем:



Что и следовало ожидать, получаем "ORA-06550: PLS-00382: выражение неправильного типа". Типы разные, и такое представление не проходит. Но вот присвоение полей этих записей между собой вполне допустимо. Пример:

SET SERVEROUTPUT ON

DECLARE

TYPE is\_SmplRecOne IS RECORD

(

m\_Fld1 VARCHAR2(10),

m\_Fld2 VARCHAR2(30),

m\_DtFld DATE,

m\_Fld3 INTEGER,

m\_Fld4 VARCHAR2(100)

);

TYPE is\_SmplRecTwo IS RECORD

(

m\_Fld1 VARCHAR2(10),

m\_Fld2 VARCHAR2(30),

m\_DtFld DATE,

m\_Fld3 INTEGER,

m\_Fld4 VARCHAR2(100)

);

MY\_SMPLONE is\_SmplRecOne;

MY\_SMPLTWO is\_SmplRecTwo;

BEGIN

MY\_SMPLONE.m\_Fld3 := 100;

MY\_SMPLONE.m\_Fld4 := 'Buber';

MY\_SMPLTWO.m\_Fld3 := MY\_SMPLONE.m\_Fld3;

MY\_SMPLTWO.m\_Fld4 := MY\_SMPLONE.m\_Fld4;

DBMS\_OUTPUT.enable;

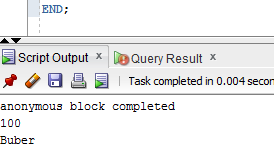
DBMS\_OUTPUT.put\_line(TO\_CHAR(MY\_SMPLTWO.m\_Fld3));

DBMS\_OUTPUT.put\_line(MY\_SMPLTWO.m\_Fld4);

END;

/

Получаем после обработки:



Теперь все правильно: типы полей совпадают и присвоение проходит нормально. Присвоить значения полям, базы можно и с помощью оператора SELECT, выбрав одну запись из таблицы. Например, давайте запишем такой блок:

DECLARE

TYPE is\_Customers IS RECORD

(

m\_COMPANY CUSTOMERS.COMPANY%TYPE,

m\_CUST\_REP CUSTOMERS.CUST\_REP%TYPE,

m\_CREDIT\_LIMIT CUSTOMERS.CREDIT\_LIMIT%TYPE

);

MY\_CUST is\_Customers;

BEGIN

SELECT COMPANY, CUST\_REP, CREDIT\_LIMIT INTO MY\_CUST

FROM CUSTOMERS WHERE CUST\_NUM = 2108;

DBMS\_OUTPUT.enable;

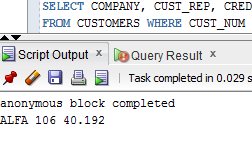
DBMS\_OUTPUT.put\_line(MY\_CUST.m\_COMPANY||' '||

TO\_CHAR(MY\_CUST.m\_CUST\_REP)||' '||TO\_CHAR(MY\_CUST.m\_CREDIT\_LIMIT));

END;

/

Здесь объявлена запись на основе типов таблицы CUSTOMERS, и выбрана одна запись из этой таблицы с помощью, оператора INTO - все это отправлено в переменные записи:



**TABLE**

Тепер рассмотрим составной тип, который является наиболее мощным и простым средством скоростной обработки больших массивов данных. Такой составной тип имеет название TABLE. Рассмотрим его синтаксис: (он подобен синтаксису RECORD)

----- TYPE - тип таблицы - IS TABLE OF тип - INDEX BY BINARY\_INTEGER ---------

В своей сути это одномерный массив скалярного типа. Он не может содержать тип RECORD или другой тип TABLE. Но может быть объявлен от любого другого стандартного типа. Также TABLE может быть объявлен от атрибута %ROWTYPE.

Тип TABLE, можно представить как одну из разновидностей коллекции. Хотя в более глубоком понимании это не совсем так. При первом рассмотрении он похож на массив языка C. Массив TABLE индексируется типом BINARY\_INTEGER (можно использовать и другие типы, но не все) и может содержать, что-то вроде - 2,147,483,647 - 0 - + 2,147,483,647 как в положительную, так и в отрицательную часть. Начинать индексацию можно с 0 или 1 или любого другого числа. Если массив будет иметь разрывы (разряженный), то это не окажет какой-либо дополнительной нагрузки на память. Так же следует помнить, что для каждого элемента, например, типа VARCHAR2 будет резервироваться столько памяти, сколько вы укажете, по этому определяйте размерность элементов по необходимости. Скажем не стоит объявлять VARCHAR2(255), если хотите хранить строки менее 100 символовю. Давайте объявим массив и посмотрим, как он работает. Запишем такой блок:

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

TYPE m\_SmplTblData IS TABLE OF DATE

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

MY\_TBL\_DT m\_SmplTblData;

BEGIN

MY\_TBL(1) := 'Buber';

MY\_TBL\_DT(2) := SYSDATE - 1;

DBMS\_OUTPUT.enable;

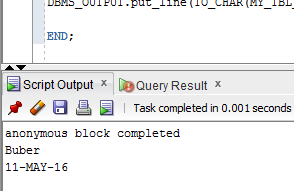
DBMS\_OUTPUT.put\_line(TO\_CHAR(MY\_TBL(1)));

DBMS\_OUTPUT.put\_line(TO\_CHAR(MY\_TBL\_DT(2)));

END;

/

Получаем после выполнения



Рассмотрим, что здесь происходит. Мы объявили две одномерные коллекции m\_SmplTable, m\_SmplTblData, одна из них содержит элементы размерностью VARCHAR2(128), другая DATE. Затем объявили две переменные данного типа - MY\_TBL, MY\_TBL\_DT и присвоили первому элементу строкового массива значение "Buber", а второму элементу массива дат, значение текущей даты минус 1. Результат вывели на консоль. При этом хорошо видно, что тип TABLE очень схож с таблицей БД и содержит обычно два ключа KEY и VALUE, ключ и значение. Ключ имеет тип BINARY\_INTEGER:

-- переменная(KEY)VALUE –

В нашем случае имеет место:

Ключ (KEY) Значение (VALUE)

MY\_TBL

1 Buber

MY\_TBL\_DT

2 01.11.03

Другой пример

DECLARE

TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);

salary\_list salary;

name VARCHAR2(20);

BEGIN

-- adding elements to the table

salary\_list('Rajnish') := 62000;

salary\_list('Minakshi') := 75000;

salary\_list('Martin') := 100000;

salary\_list('James') := 78000;

-- printing the table

name := salary\_list.FIRST;

WHILE name IS NOT null LOOP

dbms\_output.put\_line

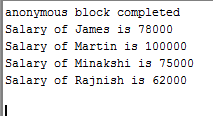
('Salary of ' || name || ' is ' || TO\_CHAR(salary\_list(name)));

name := salary\_list.NEXT(name);

END LOOP;

END;

Результат



Нужно обратить внимание на следующее:

* Число строк таблицы ни чем не ограничено. Единственное ограничение это значения, которые могут быть представлены типом индекса.
* Порядок элементов таблицы PL/SQL не обязательно должен быть строго определен. Эти элементы хранятся в памяти не подряд как массивы и по этому могут вводится с произвольными ключами.
* Ключи, используемые в таблице PL/SQL, не обязательно должны быть последовательными. В качестве индекса таблицы может быть использовано любое значение или выражение имеющее тип BINARY\_INTEGER (опять же зависит от типа).

При обращении к несуществующему элементу получите сообщение об ошибке! Давайте запишем вот такой блок:

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

BEGIN

FOR i IN 1..10 LOOP

MY\_TBL(i) := TO\_CHAR(i+5);

END LOOP;

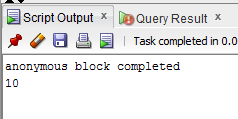
DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(TO\_CHAR(MY\_TBL(5)));

END;

/

Получаем:



Здесь мы объявили коллекцию из строковых переменных, и с помощью известного вам цикла **FOR** записали с 1-го по 10-й элемент значениями исходный плюс пять. Затем мы вывели на экран пятый элемент, как и следовало ожидать его значение равно 10 (5+5). Вот один из способов наполнения массива значениями. Но этот способ бесполезен и показан в качестве примера. А, вот следующий пример более осмыслен. Перепишем блок где мы использовали SELECT INTO:

DECLARE

TYPE is\_Customers IS TABLE OF CUSTOMERS%ROWTYPE

INDEX BY BINARY\_INTEGER;

MY\_CUST is\_Customers;

BEGIN

SELECT \* INTO MY\_CUST(100)

FROM CUSTOMERS WHERE CUST\_NUM = 2108;

DBMS\_OUTPUT.enable;

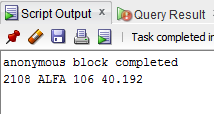
DBMS\_OUTPUT.put\_line(TO\_CHAR(MY\_CUST(100).CUST\_NUM)||' '||MY\_CUST(100).COMPANY||' '||

TO\_CHAR(MY\_CUST(100).CUST\_REP)||' '||TO\_CHAR(MY\_CUST(100).CREDIT\_LIMIT));

END;

/

Получаем:



Здесь мы объявили коллекцию с типом запись из таблицы **CUSTOMERS.** И получили тот же результат как в прошлый раз, но более элегантно. Объявляем тип:

TYPE is\_Customers IS TABLE OF CUSTOMERS%ROWTYPE

INDEX BY BINARY\_INTEGER;

Это значит, что каждая строка коллекции содержит полную запись из таблицы БД CUSTOMERS. Далее сотому элементу массива присваиваем значение записи таблицы с CUST\_NUM = 2108. Вот так это выглядит изнутри:

MY\_CUST(98)

.

MY\_CUST(100) ->

MY\_CUST(100).CUST\_NUM MY\_CUST(100).COMPANY MY\_CUST(100).CUST\_REP MY\_CUST(100).CREDIT\_LIMIT

2108 Унесенные ветром 109 55,323

.

.

MY\_CUST(n)

Обращаться к отдельному элементу (полю) записи можно через точечную нотацию.

Отдельный вид это NESTED TABLES (без INDEX BY)

Также есть коллекция VARRAY

<https://habrahabr.ru/post/254355/> а ещё лучше официальная документация

**Аттрибуты таблиц**

С таблицами PL/SQL мы познакомились, теперь разберем их атрибуты. Атрибуты таблиц имеют следующий синтаксис:

-------- таблица.атрибут ---------------

Где таблица - это ссылка на таблицу PL/SQL, а атрибут - собственно сам атрибут. Основные атрибуты таблиц PL/SQL следующие:

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Возвращаемый тип** | **Описание** |
| COUNT | NUMBER | Возвращает число строк таблицы. |
| DELETE | - | Удаляет строки таблицы. |
| EXISTS | BOOLEAN | Возвращает TRUE если указанный элемент находится в таблице иначе FALSE. |
| FIRST | BINARY\_INTEGER | Возвращает индекс первой строки таблицы. |
| LAST | BINARY\_INTEGER | Возвращает индекс последней строки таблицы. |
| NEXT | BINARY\_INTEGER | Возвращает индекс строки таблицы, которая следует за указанной строкой. |
| PRIOR | BINARY\_INTEGER | Возвращает индекс строки таблицы, которая предшествует указанной строке. |

Вот основные атрибуты таблиц PL/SQL. Теперь давайте разберем их на примерах. Начнем с атрибута COUNT. Рассмотрим блок:

-- COUNT

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

BEGIN

FOR i IN 1..10 LOOP

MY\_TBL(i) := TO\_CHAR(i+5);

END LOOP;

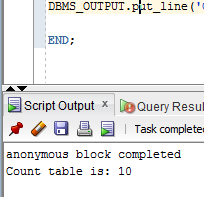
DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

END;

/

Получаем:



С помощью цикла FOR мы ввели в таблицу 10 значений и применив атрибут - COUNT получили количество записей в таблице.

Следующий атрибут DELETE. Он имеет аргументы:

* DELETE - удаляет все строки таблицы.
* DELETE(n) - удаляет n-ю строку коллекции.
* DELETE(n,m) - удаляет строки коллекции с n по m.

Рассмотрим для примера такой блок:

-- DELETE

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

BEGIN

MY\_TBL(1) := 'One';

MY\_TBL(3) := 'Three';

MY\_TBL(-2) := 'Minus Two';

MY\_TBL(0) := 'Zero';

MY\_TBL(100) := 'Hundred';

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(MY\_TBL(1));

DBMS\_OUTPUT.put\_line(MY\_TBL(3));

DBMS\_OUTPUT.put\_line(MY\_TBL(-2));

DBMS\_OUTPUT.put\_line(MY\_TBL(0));

DBMS\_OUTPUT.put\_line(MY\_TBL(100));

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

MY\_TBL.DELETE(100);

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

MY\_TBL.DELETE(1,3);

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

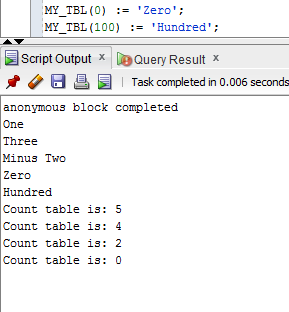
MY\_TBL.DELETE;

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

END;

/

Получаем:



В данном случае сначала мы удалили 100-ю запись коллекции, затем с 1 по 3 (при этом 2-й записи не существовало, но это не столь важно) и затем очистили всю коллекцию.Не путайте его с оператором DML DELETE. Очистить всю коллекцию целиком, можно и так:

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

MY\_TBL\_Empty m\_SmplTable;

BEGIN

MY\_TBL(1) := 'One';

MY\_TBL(3) := 'Three';

MY\_TBL(-2) := 'Minus Two';

MY\_TBL(0) := 'Zero';

MY\_TBL(100) := 'Hundred';

DBMS\_OUTPUT.enable;

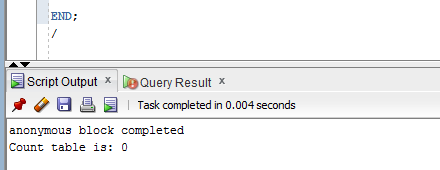
MY\_TBL := MY\_TBL\_Empty;

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

END;

/

Получаем:



Как видно, присвоение пустой коллекции (а при обьявлении коллекции она пустая) вся таблица очистилась за один заход.

Далее аттрибут EXISTS. Рассмотрим такой блок:

-- EXISTS

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

BEGIN

MY\_TBL(1) := 'Miller';

MY\_TBL(3) := 'Kolobok';

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

IF (MY\_TBL.EXISTS(1)) THEN

DBMS\_OUTPUT.put\_line(MY\_TBL(1));

ELSE

DBMS\_OUTPUT.put\_line('MY\_TBL(1) is not exist!');

END IF;

IF (MY\_TBL.EXISTS(3)) THEN

DBMS\_OUTPUT.put\_line(MY\_TBL(3));

ELSE

DBMS\_OUTPUT.put\_line('MY\_TBL(3) is not exist!');

END IF;

IF (MY\_TBL.EXISTS(2)) THEN

DBMS\_OUTPUT.put\_line(MY\_TBL(2));

ELSE

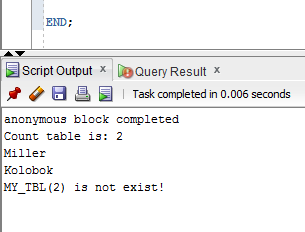
DBMS\_OUTPUT.put\_line('MY\_TBL(2) is not exist!');

END IF;

END;

/

Далее получаем:



Здесь хорошо видно, что в объявленной коллекции существуют только две строки с номером 1 и 3. А строка с номером 2 не существует.

Следующий пример показывает как работают атрибуты FIRST и LAST. Запишем такой блок:

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

FRST BINARY\_INTEGER;

LST BINARY\_INTEGER;

BEGIN

MY\_TBL(1) := 'One';

MY\_TBL(3) := 'Three';

MY\_TBL(-2) := 'Minus Two';

MY\_TBL(0) := 'Zero';

MY\_TBL(100) := 'Hundred';

DBMS\_OUTPUT.enable;

DBMS\_OUTPUT.put\_line(MY\_TBL(1));

DBMS\_OUTPUT.put\_line(MY\_TBL(3));

DBMS\_OUTPUT.put\_line(MY\_TBL(-2));

DBMS\_OUTPUT.put\_line(MY\_TBL(0));

DBMS\_OUTPUT.put\_line(MY\_TBL(100));

DBMS\_OUTPUT.put\_line('Count table is: '||TO\_CHAR(MY\_TBL.COUNT));

FRST := MY\_TBL.FIRST;

LST := MY\_TBL.LAST;

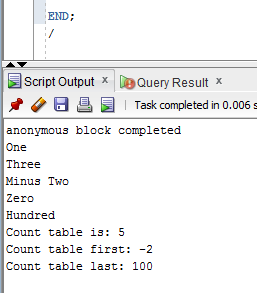
DBMS\_OUTPUT.put\_line('Count table first: '||TO\_CHAR(FRST));

DBMS\_OUTPUT.put\_line('Count table last: '||TO\_CHAR(LST));

END;

/

Результат



Хорошо видно, что количество записей в коллекции пять, и первая запись имеет индекс -2, а последняя 100. Далее, рассмотрим на примере работу атрибутов NEXT и PRIOR. С их помощью можно перемещаться по коллекции. Рассмотрим такой блок:

-- NEXT PRIOR

DECLARE

TYPE m\_SmplTable IS TABLE OF VARCHAR2(128)

INDEX BY BINARY\_INTEGER;

MY\_TBL m\_SmplTable;

m\_Cnt BINARY\_INTEGER;

a BINARY\_INTEGER;

BEGIN

FOR i IN 1..10 LOOP

MY\_TBL(i) := TO\_CHAR(i+5);

END LOOP;

DBMS\_OUTPUT.enable;

-- FOR i IN MY\_TBL.FIRST..MY\_TBL.LAST LOOP

-- DBMS\_OUTPUT.put\_line('Table is: '||TO\_CHAR(MY\_TBL(i)));

-- END LOOP;

m\_Cnt := MY\_TBL.FIRST;

LOOP

DBMS\_OUTPUT.put\_line('Table is: '||TO\_CHAR(MY\_TBL(m\_Cnt)));

EXIT WHEN m\_Cnt = MY\_TBL.LAST;

m\_Cnt := MY\_TBL.NEXT(m\_Cnt);

END LOOP;

m\_Cnt := MY\_TBL.LAST;

LOOP

DBMS\_OUTPUT.put\_line('Table is: '||TO\_CHAR(MY\_TBL(m\_Cnt)));

EXIT WHEN m\_Cnt = MY\_TBL.FIRST;

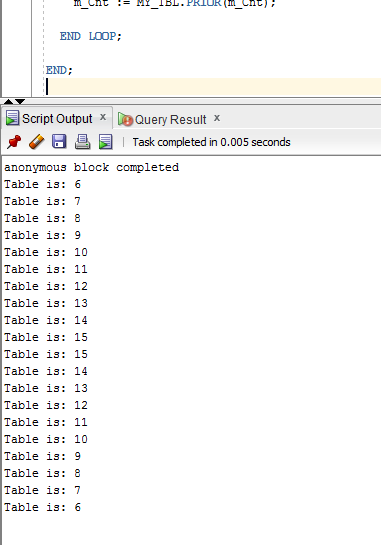
m\_Cnt := MY\_TBL.PRIOR(m\_Cnt);

END LOOP;

END;

/

После запуска получаем:



Детальнее рассмотрим, что здесь происходит. Первый цикл наполняет коллекцию с 1-го по 10-й элементы, затем первый цикл LOOP выводит элементы коллекции по возрастанию, а второй цикл LOOP выводит элементы коллекции по убыванию. Хорошо видно, как при этом срабатывают атрибуты NEXT, PRIOR, а так же FIRST, LAST. Закомментированный второй цикл FOR делает более компактный код, можете его раcкомментировать и проверить. Домашнее задание сделать третий цикл FOR, который будет выводить данные как последний LOOP.

**Неявные курсоры**

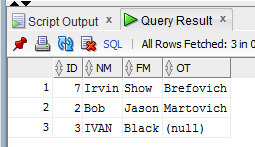
Мы с вами, уже рассмотрели основные концепции определения и работы с курсорами. Также необходимо рассмотреть отдельный вид курсоров - неявные курсоры. Оператор SELECT представляет собой в прямом определении "КУРСОР", или еще это называют SQL-курсор. Далее следует что любой оператор DML представляет из себя курсор, так как каждый оператор DML выполняется в пределах контекстной области и по этому имеет курсор указывающий на контекстную область. В отличии от явных, SQL-курсор не открывается и не закрывается. PL/SQL - сам неявно открывает SQL-курсор, обрабатывает SQL-оператор и закрывает SQL-курсор. Для SQL-курсора операторы FETCH, OPEN, CLOSE не нужны, но с ними можно использовать курсорные атрибуты, вот таким образом:

------ SQL%АТРИБУТ КУРСОРА ---------------

Давайте рассмотрим конкретный пример. Пусть нам нужно изменить поле NM в таблице PEOPLE, где поле ID содержит значение 555. Вначале посмотрим данные в таблице:

SELECT \* FROM PEOPLE

Получаем



Но строки, где бы поле ID содержало 555, нет. Это нам и нужно, для того, чтобы продемонстрировать работу атрибута. Итак, запишем блок:

BEGIN

UPDATE PEOPLE

SET NM = 'Pupkin'

WHERE ID = 555;

IF(SQL%NOTFOUND) THEN

INSERT INTO PEOPLE(ID, NM, FM, OT)

VALUES(555, 'Pupkin', 'Axlamon', 'Feodosovich');

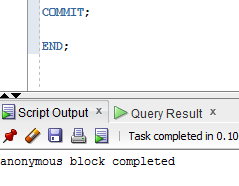
END IF;

COMMIT;

END;

/

Получаем:

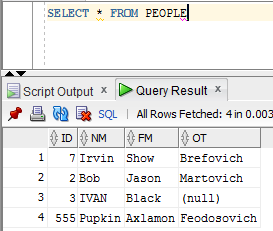


Посмотрим, что вышло:

SELECT \* FROM PEOPLE

/

Результат



Как видите, сработал атрибут SQL-курсора, %NOTFOUND. И, так как записи с таким значением поля ID не было, то с помощью оператора INSERT мы его добавили в таблицу PEOPLE. То же можно было сделать применив атрибут SQL-курсора, %ROWCOUNT. Вот так:

BEGIN

UPDATE PEOPLE

SET NM = 'Mirkin'

WHERE ID = 888;

IF(SQL%ROWCOUNT = 0) THEN

INSERT INTO PEOPLE(ID, NM, FM, OT)

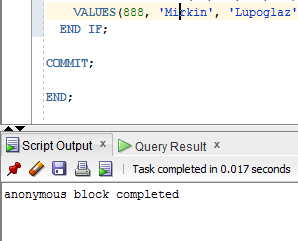
VALUES(888, 'Mirkin', 'Lupoglaz', 'Kotletovich');

END IF;

COMMIT;

END;

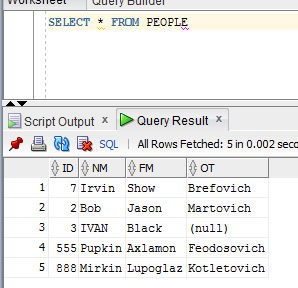
Получаем:



Смотрим содержимое таблицы PEOPLE:

SELECT \* FROM PEOPLE

/



И здесь все сработало верно, с той разницей, что мы применили атрибут SQL-курсора %ROWCOUNT. Давайте рассмотрим более сложную ситуацию, с применением атрибута SQL-курсора %NOTFOUND. Мы уже рассматривали оператор SELECT - формы SELECT ... INTO. Но, не заостряли внимание. Итак, SELECT ... INTO, это как бы некий эквивалент явного курсора с применением оператора выборки FETCH. Но сам по себе он является неявным курсором с выборкой в переменную. Если явный курсор вида:

CURSOR get\_people IS

SELECT \* FROM PEOPLE;

-- Record to store the fetched data --

v\_gt get\_people%ROWTYPE;

.

.

.

FETCH get\_people INTO v\_gt;

Выбирает данные с помощью оператора FETCH INTO в переменную v\_gt, которая в свою очередь, является одномерной коллекцией на основе курсора get\_people. То запись вида:

-- Record to store the fetched data --

v\_gt PEOPLE%ROWTYPE;

SELECT \* INTO v\_gt FROM PEOPLE;

абсолютно ей эквивалентна. SELECT ... INTO и есть неявный курсор с выборкой данных в одномерную коллекцию v\_gt, так как они обе имеют один и тот же тип определения ROWTYPE и содержат четыре переменных вида:

v\_gt.ID, v\_gt.NM, v\_gt.FM, v\_gt.OT

С той лишь разницей, что первая объявлена на основе курсора (поля которого определены собственно выражением SELECT), а вторая на основе полей таблицы. Безусловно, явные курсоры наиболее предпочтительны, так как более наглядны и применимы. Но если вам необходимо выполнить что-то очень простое, то можно использовать SELECT ... INTO. В остальных желательно использовать только явные курсоры.

Итак, собственно переходим к делу, если применить атрибут %NOTFOUND совместно с конструкцией SELECT ... INTO, то он может не сработать, так как если SELECT ... INTO не получит запись, то возникнет ошибка "ORA-1403 no data found". Для этого мы запишем вот такой блок и разберем его:

DECLARE

m\_PLP PEOPLE%ROWTYPE;

BEGIN

DBMS\_OUTPUT.enable;

SELECT \* INTO m\_PLP FROM PEOPLE

WHERE ID = 999;

IF (SQL%NOTFOUND) THEN

INSERT INTO PEOPLE(ID, NM, FM, OT)

VALUES(999, 'Volopasov', 'Lobotryas', 'Oslovich');

END IF;

COMMIT;

EXCEPTION

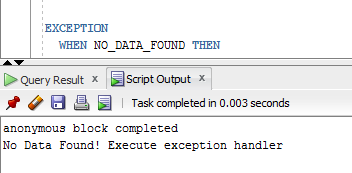
WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.put\_line('No Data Found! Execute exception handler');

END;

/

Результат



Как видите, сработал блок EXCEPTION (как с ним управляться - это чуть позже). Что и должно было произойти в нашем случае. Так как блок "SELECT \* INTO..." не получил ни одной записи, то сработал блок EXCEPTION с условием NO\_DATA\_FOUND. Вот так производится обработка атрибутов при использовании неявных курсоров.

Итак, немного разобравшись с неявными курсорами приступим вплотную к изучению явных курсоров. Если кто из вас был внимателен то заметил, что одна из строк в последнем курсоре с циклом выборки выдалась дважды. А произошло это по той причине, что оператор FETCH с одними данными сработал дважды. Домашняя работа: подумать над тем как устранить этот недочет.

Рассмотрим как писать курсоры с циклами. Для работы будем использовать таблицу OFFICES. Для начала запишем вот такой блок:

SET SERVEROUTPUT ON

DECLARE

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

v\_gt get\_offices%ROWTYPE;

BEGIN

DBMS\_OUTPUT.enable;

OPEN get\_offices;

LOOP

DBMS\_OUTPUT.put\_line('Get Data: '||TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

||v\_gt.REGION||' '||TO\_CHAR(v\_gt.MGR)||' '||TO\_CHAR(v\_gt.TARGET)||' '||TO\_CHAR(v\_gt.SALES));

FETCH get\_offices INTO v\_gt;

EXIT WHEN get\_offices%NOTFOUND;

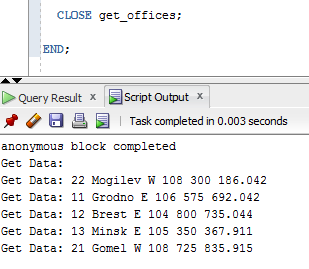
END LOOP;

CLOSE get\_offices;

END;

/

После выполнения получаем:



Используется стандартный цикл LOOP, как правило это самый распространенный подход при написании курсора. А первая строка, пустая вследствие того, что put\_line отработал с пустыми переменными. Этот недочет можно исправить, вынеся первый оператор FETCH за тело цикла вот так:

DECLARE

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

v\_gt get\_offices%ROWTYPE;

BEGIN

DBMS\_OUTPUT.enable;

OPEN get\_offices;

FETCH get\_offices INTO v\_gt;

LOOP

DBMS\_OUTPUT.put\_line('Get Data: '||TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

||v\_gt.REGION||' '||TO\_CHAR(v\_gt.MGR)||' '||TO\_CHAR(v\_gt.TARGET)||' '||TO\_CHAR(v\_gt.SALES));

FETCH get\_offices INTO v\_gt;

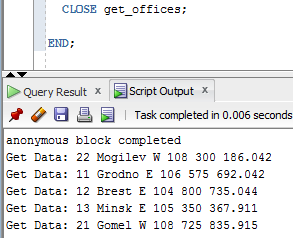
EXIT WHEN get\_offices%NOTFOUND;

END LOOP;

CLOSE get\_offices;

END;

Получаем:



Вот теперь количество строк точно соответствует количеству записей в таблице.Теперь сделаем выборку с циклом **WHILE** и посмотрим, что получится. Запишем следующий блок:

SET SERVEROUTPUT ON

DECLARE

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

v\_gt get\_offices%ROWTYPE;

BEGIN

DBMS\_OUTPUT.enable;

OPEN get\_offices;

FETCH get\_offices INTO v\_gt;

WHILE (get\_offices%FOUND) LOOP

DBMS\_OUTPUT.put\_line('Get Data: '||TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

||v\_gt.REGION||' '||TO\_CHAR(v\_gt.MGR)||' '||TO\_CHAR(v\_gt.TARGET)||' '||TO\_CHAR(v\_gt.SALES));

FETCH get\_offices INTO v\_gt;

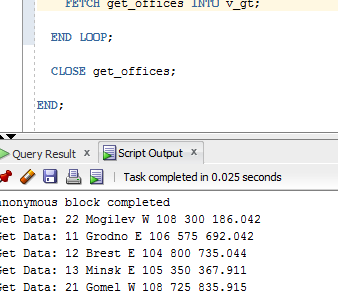
END LOOP;

CLOSE get\_offices;

END;

/

После выполнения получаем:



А, вот здесь выноска первого FETCH за тело цикла, просто необходима по определению, иначе сам цикл WHILE будет трудно заставить работать. В дальнейшем я бы рекомендовал не использовать данный цикл для работы с курсором вследствие его неудобства.

Рассмотрим выборку данных из курсоров с помощью цикла FOR. Давайте посмотрим на классический цикл выборки LOOP:

DECLARE

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

-- Объявляется курсор.

v\_gt get\_offices%ROWTYPE;

-- Объявляется переменная запись, где будут хранится данные.

BEGIN

OPEN get\_offices;

-- Открывается курсор.

-- Запускается цикл чтения.

LOOP

-- Выбираются данные.

FETCH get\_offices INTO v\_gt;

EXIT WHEN get\_offices%NOTFOUND;

END LOOP;

-- Закрывается курсор.

CLOSE get\_offices;

END;

/

А вот, теперь посмотрите на вот такой блок:

DECLARE

CURSOR get\_offices IS

SELECT \* FROM OFFICES;

BEGIN

DBMS\_OUTPUT.enable;

FOR v\_gt IN get\_offices LOOP

DBMS\_OUTPUT.put\_line('Get Data: '||TO\_CHAR(v\_gt.OFFICE)||' '||v\_gt.CITY||' '

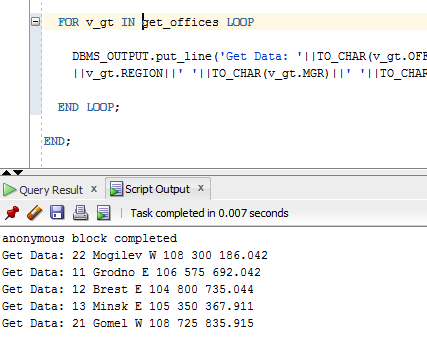
||v\_gt.REGION||' '||TO\_CHAR(v\_gt.MGR)||' '||TO\_CHAR(v\_gt.TARGET)||' '||TO\_CHAR(v\_gt.SALES));

END LOOP;

END;

/

Результат:



Действия все те же самые, но кода гораздо меньше. При работе с циклом FOR все становится на порядок проще, во-первых, открывать и закрывать курсор не нужно, он делает это сам не явно. Во-вторых, объявлять переменную запись не нужно, то есть нужно, но она определяется гораздо проще, в-третьих, меньше строчек кода. Теперь по порядку, в классическом FOR:

FOR v\_gt ( переменная запись, то же что и определение v\_gt get\_offices%ROWTYPE )

IN get\_offices ( это и есть диапазон выборки сам курсор от и до как оператор .. ) LOOP начало цикла!

Мы с вами рассматривали оператор SELECT, который получал все данные из таблиц. Но как известно, чаще вы будете использовать условие WHERE в курсорном операторе SELECT. И в дальнейшем вам будет необходимо определять условия выборки. Cамый простой способ сделать это вот так:

-- Пишем курсор с условием

CURSOR get\_sls IS

SELECT \* FROM SALESREPS

WHERE AGE = 37;

Замечательно, а если нужно другое значение отличное от 37? Что тогда? Напрашивается вывод о том, что в этом случае необходим курсор с параметрами или "параметризованный курсор". Он определяется вот так:

CURSOR get\_sls(INAGE NUMBER) IS

SELECT \* FROM SALESREPS

WHERE AGE = INAGE;

Но еще более правильно, будет сделать вот так:

CURSOR get\_sls(INAGE SALESREPS.AGE%TYPE) IS

SELECT \* FROM SALESREPS

WHERE AGE = INAGE;

Применив оператор TYPE мы сделали код более мобильным, так как если тип поля изменится, то не нужно будет проверять весь код в поисках ошибки. Теперь при открытии такого курсора, его оператор OPEN будет принимать передаваемый параметр вот так:

OPEN get\_sls(37);

Подытожим наши знания следующим блоком:

SET SERVEROUTPUT ON

DECLARE

CURSOR get\_sls(INAGE SALESREPS.AGE%TYPE) IS

SELECT \* FROM SALESREPS

WHERE AGE = INAGE;

in\_sls get\_sls%ROWTYPE;

BEGIN

DBMS\_OUTPUT.enable;

OPEN get\_sls(37);

FETCH get\_sls INTO in\_sls;

LOOP

DBMS\_OUTPUT.put\_line('Record is: '||in\_sls.NAME||' '||in\_sls.TITLE||' '||in\_sls.HIRE\_DATE);

FETCH get\_sls INTO in\_sls;

EXIT WHEN get\_sls%NOTFOUND;

END LOOP;

CLOSE get\_sls;

END;

/

CREATE TABLE OFFICES

(

OFFICE INTEGER PRIMARY KEY, OFFICE,CITY,REGION,MGR,TARGET,SALES

CITY VARCHAR2(30) NOT NULL,

REGION VARCHAR2(30) NOT NULL,

MGR INTEGER,

TARGET NUMBER,

SALES NUMBER NOT NULL

);

insert into offices(OFFICE,CITY,REGION,MGR,TARGET,SALES) values(22,'Mogilev','W',108,300,186.042);

insert into offices(OFFICE,CITY,REGION,MGR,TARGET,SALES) values(11,'Grodno','E',106,575,692.042);

insert into offices(OFFICE,CITY,REGION,MGR,TARGET,SALES) values(12,'Brest','E',104,800,735.044);

insert into offices(OFFICE,CITY,REGION,MGR,TARGET,SALES) values(13,'Minsk','E',105,350,367.911);

insert into offices(OFFICE,CITY,REGION,MGR,TARGET,SALES) values(21,'Gomel','W',108,725,835.915);

commit;

create table CUSTOMERS(

CUST\_NUM INTEGER,

COMPANY VARCHAR(30),

CUST\_REP INTEGER,

CREDIT\_LIMIT NUMBER

);

insert into CUSTOMERS(CUST\_NUM,COMPANY,CUST\_REP,CREDIT\_LIMIT) values (2111,'BigBill',101,50.765);

insert into CUSTOMERS(CUST\_NUM,COMPANY,CUST\_REP,CREDIT\_LIMIT) values (2102,'Krinica',102,65.345);

insert into CUSTOMERS(CUST\_NUM,COMPANY,CUST\_REP,CREDIT\_LIMIT) values (2103,'IBA',103,49.927);

insert into CUSTOMERS(CUST\_NUM,COMPANY,CUST\_REP,CREDIT\_LIMIT) values (2123,'BSB',104,98.120);

insert into CUSTOMERS(CUST\_NUM,COMPANY,CUST\_REP,CREDIT\_LIMIT) values (2101,'BVEB',105,22.109);

insert into CUSTOMERS(CUST\_NUM,COMPANY,CUST\_REP,CREDIT\_LIMIT) values (2108,'ALFA',106,40.192);

commit;

CREATE TABLE PEOPLE

(

ID NUMBER PRIMARY KEY,

NM VARCHAR2(50),

FM VARCHAR2(50),

OT VARCHAR2(50)

);

INSERT INTO PEOPLE(ID, NM, FM, OT) VALUES(7, 'Irvin', 'Show', 'Brefovich');

INSERT INTO PEOPLE(ID, NM, FM, OT) VALUES(2, 'Bob', 'Jason', 'Martovich');

INSERT INTO PEOPLE(ID, NM, FM, OT) VALUES(3, 'IVAN', 'Black', NULL);

commit;