Использование процедур и функций. Требования к оформлению процедур и функций. Объединение процедур и функций в пакеты Использование SQL в PL-SQL, динамических SQL Использования статического и динамического sql в блоках PLSQL. Переключение контекстов. Коллекции Пользовательские типы данных. Использование коллекций.

Типы специальных символов **PL/SQL**:

1. Арифметические операторы:

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

2. 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

3. 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

4. Комментарии и метки

--	Комментарий в одной строке

/*	Начало многострочного комментария
***/**	Конец многострочного комментария
>>	Начало метки
<<	Конец метки

Самое интересное, нафига, такое количество способов, сказать не равно! Хотя **!=** по моему вполне достаточно! Я например больше люблю вот так **<>**! :) Но вообще кому как нравится! Так же замечу, что вложенные комментарии не допускаются! Проще использовать **/* */**как в языке **C**! А теперь про самое интересное - блоки **PL/SQL**.

Блоки **PL/SQL**, могут быть, как я уже говорил "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;

/ -- символ завершения для запуска блока на компиляцию
```

. 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. Что ж, пришло время вспомнить старого знакомого! Работа с **NULL**. Оператор **IF** работает с **NULL** достаточно просто и эффективно. Если записать:

```
IF(X = NULL) THEN        -- проверка условия
  .
  .
END IF;   -- конец условного оператора.

IF(X != NULL) THEN       -- проверка условия
  .
  .
END IF;   -- конец условного оператора.
```

Такое 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.
```

От троичной логики никуда не деться, по этому запоминайте хорошенько! :) 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 по моему опыту 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.
```

Основные типы данных, используемые в PL/SQL:

- **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

Задание:

Напишите программный код, который бы:

- принимал от пользователя в качестве первого параметра букву английского алфавита;
- принимал от пользователя в качестве второго параметра число (количество приращений);
- на основе введенной информации выводил бы последовательно буквы по алфавиту, начиная с введенной пользователем. Количество выводимых букв должно соответствовать количеству приращений, принятых от пользователя.

Пример работы программы должен выглядеть так, как представлено на рис. Лаб. 15.3-1.

```
Введите значение для буква: A
прежний   2: sLetter nvarchar2(1) := '&Буква';
новый   2: sLetter nvarchar2(1) := 'A';
Введите значение для количество_приращений: 10
прежний   3: nNumber number(4,0) := &Количество_приращений;
новый   3: nNumber number(4,0) := 10;
A
B
C
D
E
F
G
H
I
J
K

Процедура PL/SQL успешно завершена.
```

Рис. Лаб. 15.3-1

Примечание. Достаточно будет решить задачу только для английского алфавита.

Решение:

Соответствующий код PL/SQL должен выглядеть так (выполняется в SQL*Plus):

```
SET SERVEROUTPUT ON;

DECLARE

sLetter nvarchar2(1) := '&Буква';

nNumber number(4,0) := &Количество_приращений;

nLetterCode number(10,0) :=0;

iNumber number(4,0) := 0;

BEGIN

nLetterCode := ASCII(sLetter);

dbms_output.PUT_LINE(Chr(nLetterCode));

LOOP

nLetterCode := nLetterCode + 1;

dbms_output.PUT_LINE(Chr(nLetterCode));

iNumber := iNumber + 1;

IF iNumber >= nNumber Then

EXIT;

END IF;

END LOOP;

END;
```

4.3. Явные курсоры в коде Oracle PL/SQL, ключевые слова CURSOR, OPEN, FETCH, CLOSE

Основной базис **PL/SQL**, как вы уже наверное заметили, мы с вами, можно сказать, получили. Пришло время начинать разбираться с краеугольным, так сказать, камнем языка **PL/SQL** - а, именно с таким понятием как "КУРСОР". Но, для начала давайте, рассмотрим несколько понятий, которые предваряют этот материал. Ранее я часто упоминал, такое слово как "транзакция". Мы еще не раз к ней будем возвращаться, так как это довольно объемная тема. Но, чтобы было проще для понимания, начнем вот с чего. Допустим, многие из Вас ходили в Сбербанк платить за квартиру, за талоны техосмотра, ну и еще много чего. По своей сути вы выполняли банковскую операцию платежа. В принципе это и есть не что иное, как транзакция (банковская в данном случае). Вы отдаете квитанцию девушке операционистке, то есть вы инициируете начало транзакции, девушка проделывает пляшущее движение пальчиками по клавиатуре то есть вводит в машину ваш код платежа, сумму и т.д. тем самым выполняет операцию отъема у вас энной суммы денег! :) Затем, мило улыбаясь, говорит сколько вы должны выложить денежек и так же мило улыбаясь, берет их у вас и отсчитывает сдачу, если таковая должна иметь место. То есть происходит процесс обмена, данными (в нашем случае квитанция - деньги). Затем она быстренько разбрасывает отполовиненные квитанции по ячейкам, а вторые половинки отдает вам. Все, транзакция завершена, оплата прошла. Примерно тоже самое происходит внутри сервера **Oracle** когда вы подключаетесь к нему скажем при помощи **SQL*Plus**. К чему я все это тут вам рассказываю, а к тому что - понятие механизма обмена данными между клиентом и сервером БД, достаточно хорошо просматривается на примере работы курсоров. По мере их изучения, мы понемногу подберемся к теме, определения экземпляров БД и еще многим, достаточно сложным понятиям. Пока это описание

банковского платежа, пусть отложится у вас в памяти, оно еще нам потребуется! :) Проводя аналогии с реальными ситуациями, иногда достаточно просто разобраться со сложными процессами, которые происходят внутри БД, что я вам еще не раз продемонстрирую. Итак, в прошлый раз мы изучили оператор **%TYPE**. Надеюсь, вы поняли, что это за оператор и что он позволяет выполнять. Так же в **PL/SQL** для удобства работы с данными имеется еще один очень полезный и наиболее часто применимый оператор **%ROWTYPE**! Он способен проделывать, очень интересный фокус. 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**! Теперь надеюсь ясно, 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. Так же 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.

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 -----------
```

Вот так определяется явный курсор. Давайте запишем несколько определений курсоров, и вы посмотрите как это делается. Если пока что-то не понятно можете не расстраиваться, я все поясню по ходу.

Итак:

```
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;
```

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 я определил простой не параметризированный курсор, который тем не менее, получает условие через объявленную переменную, скажу сразу это не очень хороший стиль работы, но пока пусть будет так. Что же происходит когда курсор открывается? А вот, что:

1. Анализируется значение переменных привязки.

2. На основе значений переменных привязки определяется активный набор.

3. Указатель активного набора устанавливается на первую строку.

Вот таким образом происходит открытие курсора. 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**). Он разработан специально для обеспечения целостности базы данных. 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**! А, 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. Как я понимаю, вы уже поняли, что оператор **CLOSE** закрывает курсор. Давайте немного подробнее остановимся на нем. Его синтаксис таков:

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

Я уже упоминал, что хорошим стилем программирования является закрытие курсора, когда работа с ним окончена и вот почему. 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
(неверный курсор)
```

или:

```
ORA-1002 Fetch out of  Sequence
(непоследовательное считывание)
```

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: (на самом деле в вашей учебной базе таблица **OFFICES** содержит 5ть записей и больше полей, но пока просто включите свою фантазию!) :)

```
Таблица OFFICES

 OFFICE CITY
------- --------------------
     22 Запиндрищинск
     11 Красный Мотоцикл
```

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;
/
```

Теперь давайте 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. Надеюсь, что дополнительно пояснять не нужно, так как все достаточно хорошо видно в таблице!

%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**, как вы уже догадались, является полной противоположностью **%FOUND** и так же хорошо понятен из приведенной ниже таблицы:

%NOTFOUND

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:

%ISOPEN

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:

%ROWCOUNT

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.

Надеюсь теперь вам стало окончательно ясно, как работает последний курсор из прошлого шага, где мы применили атрибут **%NOTFOUND** в теле цикла, дабы оповестить оного, что пора заканчивать чтение строк! :) Что ж, применим наши знания на практике, так как сухая теория это еще не все. Перепишем, наш учебный курсор вот так:

```
SET SERVEROUTPUT ON

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:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3       -- Cursor declaration --
  4       CURSOR get_offices IS
  5              SELECT * FROM OFFICES;
  6       -- Record to store the fetched data --
  7       v_gt get_offices%ROWTYPE;
  8
  9 BEGIN
 10
 11       DBMS_OUTPUT.enable;
 12
 13       IF(get_offices%ISOPEN) THEN
 14       DBMS_OUTPUT.put_line('Cursor get_offices is open
now!');
 15       ELSE
 16       DBMS_OUTPUT.put_line('Cursor get_offices is close
now!');
 17       END IF;
 18
 19       -- location (1) -- cursor is open --
 20       OPEN get_offices;
 21
 22       IF(get_offices%ISOPEN) THEN
 23       DBMS_OUTPUT.put_line('Cursor get_offices is open
now!');
 24       ELSE
 25       DBMS_OUTPUT.put_line('Cursor get_offices is close
now!');
 26       END IF;
 27
 28       -- location (2)
 29       FETCH get_offices INTO v_gt;   -- fetch first row --
 30       IF(get_offices%FOUND) THEN
 31       DBMS_OUTPUT.put_line('Row is Found!');
 32       ELSE
 33       DBMS_OUTPUT.put_line('Row is not Found!');
 34       END IF;
 35       DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
 36              ||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));
 37
 38       -- location (3)
 39       FETCH get_offices INTO v_gt;   -- fetch second row --
 40
 41       IF(get_offices%NOTFOUND) THEN
 42       DBMS_OUTPUT.put_line('Row is not Found!');
 43       ELSE
 44       DBMS_OUTPUT.put_line('Row is Found!');
 45       END IF;
 46       DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
 47              ||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));
 48
 49       -- location (4)
 50       FETCH get_offices INTO v_gt;   -- Third first --
 51       IF(get_offices%FOUND) THEN
```

```
 52       DBMS_OUTPUT.put_line('Row is Found!');
 53       ELSE
 54       DBMS_OUTPUT.put_line('Row is not Found!');
 55       END IF;
 56       DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
 57              ||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));
 58
 59       -- location (5)
 60       CLOSE get_offices;     -- cursor is close --
 61       -- location (6)
 62
 63 END;
 64 /
```

. 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**. Как можно было догадаться, предыдущий пример, очень напоминает своей структурой, нашу учебную таблицу **CUSTOMERS**. 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] ); -------
```

Вот таким образом объявляется составной тип **RECORD**. 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:

```
SET SERVEROUTPUT ON
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 прогона в **SQL*Plus** получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2
  3  TYPE is_SmplRec IS RECORD
  4       (
  5       m_Fld1 VARCHAR2(10),
  6       m_Fld2 VARCHAR2(30)  := 'Buber',
  7       m_DtFld DATE,
  8       m_Fld3 INTEGER := 1000,
  9       m_Fld4 VARCHAR2(100) NOT NULL := 'System'
 10       );
 11
 12
 13  MY_SMPL is_SmplRec;
 14
 15  BEGIN
 16
 17  DBMS_OUTPUT.enable;
 18  DBMS_OUTPUT.put_line(MY_SMPL.m_Fld2||' '||MY_SMPL.m_Fld4);
 19
 20  END;
 21  /
Buber System

Процедура PL/SQL успешно завершена.
```

Как видно 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! Давайте так же 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:

```
SET SERVEROUTPUT ON
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_Fld4);

END;
/
```

После обработки получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
```

```
SQL> DECLARE
  2
  3  TYPE is_SmplRec IS RECORD
  4       (
  5       m_Fld1 VARCHAR2(10),
  6       m_Fld2 VARCHAR2(30)  := 'Buber',
  7       m_DtFld DATE,
  8       m_Fld3 INTEGER := 1000,
  9       m_Fld4 VARCHAR2(100) NOT NULL := 'System'
 10       );
 11
 12
 13  MY_SMPL is_SmplRec;
 14
 15  BEGIN
 16
 17  MY_SMPL.m_DtFld := SYSDATE;
 18  MY_SMPL.m_Fld4 := 'Unknown';
 19
 20  DBMS_OUTPUT.enable;
 21  DBMS_OUTPUT.put_line(MY_SMPL.m_Fld2||' '||MY_SMPL.m_Fld4);
 22
 23  END;
 24  /
Buber Unknown

Процедура PL/SQL успешно завершена.
```

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 давайте рассмотрим, как производится присваивание записей. В языке **PL/SQL** при присвоении значения одной записи другой используется, так называемая семантика копирования. Хотя 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:

```
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 := MY_SMPLTWO;

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3  TYPE is_SmplRecOne IS RECORD
  4       (
  5       m_Fld1 VARCHAR2(10),
  6       m_Fld2 VARCHAR2(30),
  7       m_DtFld DATE,
  8       m_Fld3 INTEGER,
  9       m_Fld4 VARCHAR2(100)
 10       );
 11
 12  TYPE is_SmplRecTwo IS RECORD
 13       (
 14       m_Fld1 VARCHAR2(10),
 15       m_Fld2 VARCHAR2(30),
 16       m_DtFld DATE,
 17       m_Fld3 INTEGER,
 18       m_Fld4 VARCHAR2(100)
 19       );
 20
 21
 22  MY_SMPLONE is_SmplRecOne;
 23  MY_SMPLTWO is_SmplRecTwo;
 24
 25  BEGIN
 26
 27  MY_SMPLONE := MY_SMPLTWO;
 28
 29  END;
 30  /
MY_SMPLONE := MY_SMPLTWO;
              *
ошибка в строке 27:
ORA-06550: Строка 27, столбец 15:
PLS-00382: выражение неправильного типа
ORA-06550: Строка 27, столбец 1:
PL/SQL: Statement ignored
```

Что и следовало ожидать, получаем "PLS-00382: выражение неправильного типа"! Все верно, типы то разные! И такое представление не проходит! Но вот присвоение полей этих записей между собой вполне приемлимо! Вот так:

```
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:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3  TYPE is_SmplRecOne IS RECORD
  4       (
  5       m_Fld1 VARCHAR2(10),
  6       m_Fld2 VARCHAR2(30),
  7       m_DtFld DATE,
  8       m_Fld3 INTEGER,
  9       m_Fld4 VARCHAR2(100)
 10       );
 11
 12  TYPE is_SmplRecTwo IS RECORD
 13       (
 14       m_Fld1 VARCHAR2(10),
 15       m_Fld2 VARCHAR2(30),
 16       m_DtFld DATE,
 17       m_Fld3 INTEGER,
 18       m_Fld4 VARCHAR2(100)
 19       );
 20
 21
 22  MY_SMPLONE is_SmplRecOne;
 23  MY_SMPLTWO is_SmplRecTwo;
 24
 25  BEGIN
 26
 27  MY_SMPLONE.m_Fld3 := 100;
 28  MY_SMPLONE.m_Fld4 := 'Buber';
 29
 30  MY_SMPLTWO.m_Fld3 := MY_SMPLONE.m_Fld3;
```

```
 31  MY_SMPLTWO.m_Fld4 := MY_SMPLONE.m_Fld4;
 32
 33  DBMS_OUTPUT.enable;
 34  DBMS_OUTPUT.put_line(TO_CHAR(MY_SMPLTWO.m_Fld3));
 35  DBMS_OUTPUT.put_line(MY_SMPLTWO.m_Fld4);
 36
 37  END;
 38  /
100
Buber

Процедура PL/SQL успешно завершена.
```

Вот 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:

```
SQL> DECLARE
  2
  3  TYPE is_Customers IS RECORD
  4       (
  5       m_COMPANY CUSTOMERS.COMPANY%TYPE,
  6       m_CUST_REP CUSTOMERS.CUST_REP%TYPE,
  7       m_CREDIT_LIMIT CUSTOMERS.CREDIT_LIMIT%TYPE
  8       );
  9
 10  MY_CUST is_Customers;
 11
 12  BEGIN
 13
 14  SELECT COMPANY, CUST_REP, CREDIT_LIMIT INTO MY_CUST
 15  FROM CUSTOMERS WHERE CUST_NUM = 2108;
 16
 17  DBMS_OUTPUT.enable;
 18  DBMS_OUTPUT.put_line(MY_CUST.m_COMPANY||' '||
```

```
 19  TO_CHAR(MY_CUST.m_CUST_REP)||'
'||TO_CHAR(MY_CUST.m_CREDIT_LIMIT));
 20
 21  END;
 22  /
----- TYPE - тип таблицы - IS TABLE OF тип - INDEX BY BINARY_INTEGER --
-------
```

Вот так просто объявить составной тип **TABLE**! 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. И это еще не все! **TABLE** может быть объявлен от атрибута **%ROWTYPE**! Вот где скрывается, по истине огромная мощь, этого типа данных! Но пока, все по порядку. 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** 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, 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 символов! :) Итак, давайте объявим массив и посмотрим, как он работает. Запишем такой блок:

```
SET SERVEROUTPUT ON
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 обработки в **SQL*Plus**:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2
  3       TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  4       INDEX BY BINARY_INTEGER;
  5
  6       TYPE m_SmplTblData IS TABLE OF DATE
  7       INDEX BY BINARY_INTEGER;
  8
  9
 10       MY_TBL m_SmplTable;
 11       MY_TBL_DT m_SmplTblData;
 12
 13  BEGIN
 14
```

```
 15  MY_TBL(1) := 'Buber';
 16  MY_TBL_DT(2) := SYSDATE - 1;
 17
 18  DBMS_OUTPUT.enable;
 19  DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL(1)));
 20  DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL_DT(2)));
 21
 22  END;
 23  /
Buber
01.11.03

Процедура PL/SQL успешно завершена.
```

Рассмотрим, что же 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! Вот собственно и все. 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
```

При этом можете обратить внимание на следующее:

1. Число строк таблицы ни чем не ограничено. Единственное ограничение это значения, которые могут быть представлены типом **BINARY_INTEGER**.

2. 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.

3. 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**.

Вот такие правила действуют при определении и работе с таблицами **PL/SQL** (не путать с таблицами БД!!!). Так же при обращении к несуществующему элементу получите сообщение об ошибке! Давайте запишем вот такой блок:

```
SET SERVEROUTPUT ON
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:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2
  3       TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  4       INDEX BY BINARY_INTEGER;
  5
  6       MY_TBL m_SmplTable;
  7
  8  BEGIN
  9
 10  FOR i IN 1..10 LOOP
 11  MY_TBL(i) := TO_CHAR(i+5);
 12  END LOOP;
 13
 14  DBMS_OUTPUT.enable;
 15  DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL(5)));
 16
 17  END;
 18  /
10

Процедура PL/SQL успешно завершена.
```

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 из предыдущего шага вот так:

```
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:

```
SQL> DECLARE
  2
  3  TYPE is_Customers IS TABLE OF CUSTOMERS%ROWTYPE
  4       INDEX BY BINARY_INTEGER;
```

```
  5
  6  MY_CUST is_Customers;
  7
  8  BEGIN
  9
 10  SELECT * INTO MY_CUST(100)
 11  FROM CUSTOMERS WHERE CUST_NUM = 2108;
 12
 13  DBMS_OUTPUT.enable;
 14  DBMS_OUTPUT.put_line(TO_CHAR(MY_CUST(100).CUST_NUM)||'
'||MY_CUST(100).COMPANY||' '||
 15  TO_CHAR(MY_CUST(100).CUST_REP)||'
'||TO_CHAR(MY_CUST(100).CREDIT_LIMIT));
 16
 17  END;
 18  /
2108 Унесенные ветром 109 55,323

Процедура PL/SQL успешно завершена.
```

Здесь мы объявили коллекцию, с типом запись из таблицы **CUSTOMERS**! И получили тот же результат как в прошлый раз, но более элегантно! Итак, 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**. Замечательно. Далее 100 элементу массива присваиваем значение записи таблицы с индексом 2108. Вот так это выглядит изнутри:

```
MY_CUST(98)
MY_CUST(99)
.
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)
-------- 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 все на примерах. Начнем по порядку. Атрибут **COUNT**. Запишем вот такой блок:

```
SET SERVEROUTPUT ON
-- 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:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2
  3       TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  4       INDEX BY BINARY_INTEGER;
  5
  6       MY_TBL m_SmplTable;
  7
  8  BEGIN
  9
 10  FOR i IN 1..10 LOOP
 11  MY_TBL(i) := TO_CHAR(i+5);
 12  END LOOP;
 13
 14  DBMS_OUTPUT.enable;
 15  DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
 16
 17  END;
 18  /
Count table is: 10

Процедура PL/SQL успешно завершена.
```

С помощью цикла **FOR** мы ввели в таблицу 10 значений и применив атрибут - **COUNT** получили количество записей в таблице. Вот собственно и все. Следующий атрибут **DELETE**. Тут чуть сложнее. Во-первых, 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:

```
SET SERVEROUTPUT ON
```

```
-- 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:

```
SQL> SET SERVEROUTPUT ON
SQL> -- DELETE
SQL> DECLARE
  2
  3       TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  4       INDEX BY BINARY_INTEGER;
  5
  6       MY_TBL m_SmplTable;
  7
  8  BEGIN
  9
 10  MY_TBL(1) := 'One';
 11  MY_TBL(3) := 'Three';
 12  MY_TBL(-2) := 'Minus Two';
 13  MY_TBL(0) := 'Zero';
 14  MY_TBL(100) := 'Hundred';
 15
 16  DBMS_OUTPUT.enable;
 17
 18  DBMS_OUTPUT.put_line(MY_TBL(1));
 19  DBMS_OUTPUT.put_line(MY_TBL(3));
 20  DBMS_OUTPUT.put_line(MY_TBL(-2));
 21  DBMS_OUTPUT.put_line(MY_TBL(0));
 22  DBMS_OUTPUT.put_line(MY_TBL(100));
 23  DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
 24
```

```
 25  MY_TBL.DELETE(100);
 26  DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
 27  MY_TBL.DELETE(1,3);
 28  DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
 29  MY_TBL.DELETE;
 30  DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
 31
 32  END;
 33  /
One
Three
Minus Two
Zero
Hundred
Count table is: 5
Count table is: 4
Count table is: 2
Count table is: 0

Процедура PL/SQL успешно завершена.
```

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. Вот так работает атрибут **DELETE**. И не путайте его с оператором **DML DELETE**! К стати очистить всю коллекцию целиком, можно и так:

```
SET SERVEROUTPUT ON
-- DELETE
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:

```
SQL> SET SERVEROUTPUT ON
SQL> -- DELETE
SQL> DECLARE
  2
  3       TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  4       INDEX BY BINARY_INTEGER;
  5
  6       MY_TBL m_SmplTable;
```

```
  7       MY_TBL_Empty m_SmplTable;
  8
  9  BEGIN
 10
 11  MY_TBL(1) := 'One';
 12  MY_TBL(3) := 'Three';
 13  MY_TBL(-2) := 'Minus Two';
 14  MY_TBL(0) := 'Zero';
 15  MY_TBL(100) := 'Hundred';
 16
 17  DBMS_OUTPUT.enable;
 18
 19       MY_TBL := MY_TBL_Empty;
 20
 21       DBMS_OUTPUT.put_line('Count table is:
'||TO_CHAR(MY_TBL.COUNT));
 22
 23  END;
 24  /
Count table is: 0

Процедура PL/SQL успешно завершена.
```

Как видно, присвоение пустой коллекции, а при обьявлении коллекции это именно так, вся таблица очистилась, за один, заход! Но как это делать на практике судить Вам! :) И последний на этот раз атрибут **EXISTS**. Рассмотрим такой блок:

```
SET SERVEROUTPUT ON
-- 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:

```
SQL> SET SERVEROUTPUT ON
SQL> -- EXISTS
SQL> DECLARE
  2
  3      TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  4      INDEX BY BINARY_INTEGER;
  5
  6      MY_TBL m_SmplTable;
  7
  8  BEGIN
  9
 10  MY_TBL(1) := 'Miller';
 11  MY_TBL(3) := 'Kolobok';
 12
 13  DBMS_OUTPUT.enable;
 14
 15  DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
 16
 17  IF (MY_TBL.EXISTS(1)) THEN
 18  DBMS_OUTPUT.put_line(MY_TBL(1));
 19  ELSE
 20  DBMS_OUTPUT.put_line('MY_TBL(1) is not exist!');
 21  END IF;
 22
 23  IF (MY_TBL.EXISTS(3)) THEN
 24  DBMS_OUTPUT.put_line(MY_TBL(3));
 25  ELSE
 26  DBMS_OUTPUT.put_line('MY_TBL(3) is not exist!');
 27  END IF;
 28
 29  IF (MY_TBL.EXISTS(2)) THEN
 30  DBMS_OUTPUT.put_line(MY_TBL(2));
 31  ELSE
 32  DBMS_OUTPUT.put_line('MY_TBL(2) is not exist!');
 33  END IF;
 34
 35  END;
 36  /
Count table is: 2
Miller
Kolobok
MY_TBL(2) is not exist!
```

любой оператор **DML** объемлит собой 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. В шаге 48 мы закончили работу с табличками **PEOPLE** и **OLD_PEOPLE**, если вы их удалили, то ничего страшного, просто вернитесь к шагу 46 и снова их создайте. Если они у вас остались, еще лучше! Итак, рассмотрим такой пример на основе таблички **PEOPLE**. Пусть нам нужно изменить поле **NM** в таблице **PEOPLE**, где поле **ID** содержит значение 555. Введем такой запрос, перед тем как:

```
SELECT * FROM PEOPLE
/
```

Получаем (у меня так, у вас может отличаться, если таблица удалялась или вы делали что-то еще):

```
SQL> SELECT * FROM PEOPLE
  2  /

  ID NM     FM         OT
---- ------ ---------- ---------------
   7 Irvin  Show       Brefovich
   2 Bob    Jason      Martovich
   3 IVAN   Black      NULL
```

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:

```
SQL> BEGIN
  2
  3  UPDATE PEOPLE
  4       SET NM = 'Pupkin'
  5  WHERE ID = 555;
  6
  7       IF(SQL%NOTFOUND) THEN
  8       INSERT INTO PEOPLE(ID, NM, FM, OT)
  9              VALUES(555, 'Pupkin', 'Axlamon', 'Feodosovich');
 10       END IF;
 11
 12  COMMIT;
 13
 14  END;
 15  /

Процедура PL/SQL успешно завершена.
```

Посмотрим, что вышло:

```
SELECT * FROM PEOPLE
/
```

Получаем:

```
SQL> SELECT * FROM PEOPLE
  2  /
```

```
  ID NM     FM         OT
---- ------ ---------- ---------------
   7 Irvin  Show       Brefovich
   2 Bob    Jason      Martovich
   3 IVAN   Black      NULL
 555 Pupkin Axlamon    Feodosovich
```

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**! Все получилось верно! 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:

```
SQL> BEGIN
  2
  3  UPDATE PEOPLE
  4       SET NM = 'Mirkin'
  5  WHERE ID = 888;
  6
  7       IF(SQL%ROWCOUNT = 0) THEN
  8       INSERT INTO PEOPLE(ID, NM, FM, OT)
  9              VALUES(888, 'Mirkin', 'Lupoglaz', 'Kotletovich');
 10       END IF;
 11
 12  COMMIT;
 13
 14  END;
 15  /

Процедура PL/SQL успешно завершена.
```

Смотрим содержимое таблицы **PEOPLE**:

```
SELECT * FROM PEOPLE
/
```

Получаем:

```
SQL> SELECT * FROM PEOPLE
  2  /

  ID NM     FM         OT
---- ------ ---------- ---------------
   7 Irvin  Show       Brefovich
   2 Bob    Jason      Martovich
```

```
  3 IVAN   Black      NULL
555 Pupkin Axlamon    Feodosovich
888 Mirkin Lupoglaz   Kotletovich
```

И здесь все сработало верно, с той разницей, что мы применили атрибут **SQL**-курсора **%ROWCOUNT**. Теперь, давайте рассмотрим более сложную ситуацию, с применением атрибута **SQL**-курсора **%NOTFOUND**. А начнем, вот с чего. Я уже показывал вам оператор**SELECT** - формы **SELECT ... INTO**. Но, не заострял внимание. Давайте немного отвлечемся и я все постараюсь объяснить. 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. Понятно? Если нет, идем дальше. 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;
```

Выбегает данные с помощью оператора **FETCH**, имеющего конструкцию, **INTO** в переменную **v_gt**. В свою очередь, являющейся 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! Теперь, я надеюсь, что все неясности по поводу курсоров как явных, так и не явных у вас отпали сами собой. 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 случаях 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:

```
SET SERVEROUTPUT ON

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;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3       m_PLP PEOPLE%ROWTYPE;
  4
  5  BEGIN
  6
  7       DBMS_OUTPUT.enable;
  8
  9       SELECT * INTO m_PLP FROM PEOPLE
 10       WHERE ID = 999;
 11
 12       IF (SQL%NOTFOUND) THEN
 13       INSERT INTO PEOPLE(ID, NM, FM, OT)
 14              VALUES(999, 'Volopasov', 'Lobotryas', 'Oslovich');
 15       END IF;
 16
 17       COMMIT;
 18
 19  EXCEPTION
 20       WHEN NO_DATA_FOUND THEN
 21       DBMS_OUTPUT.put_line('No Data Found! Execute exception
handler');
 22
 23  END;
 24  /
No Data Found! Execute exception handler
-- 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 лопатить весь код в поисках ошибки! Это так же считается наиболее правильным стилем

программирования! :) И теперь при открытии такого курсора, его оператор **OPEN** будет принимать передаваемый параметр вот так:

```
OPEN get_sls(37);
```

Подытожим наши заключения 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;
/
```

После запуска получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2
  3       CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS
  4              SELECT * FROM SALESREPS
  5              WHERE AGE = INAGE;
  6
  7       in_sls get_sls%ROWTYPE;
  8
  9  BEGIN
 10
 11       DBMS_OUTPUT.enable;
 12
 13       OPEN get_sls(37);
 14
 15       FETCH get_sls INTO in_sls;
 16
 17       LOOP
 18
```

```
 19              DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||'
'||in_sls.TITLE||' '||in_sls.HIRE_DATE);
 20
 21              FETCH get_sls INTO in_sls;
 22              EXIT WHEN get_sls%NOTFOUND;
 23
 24      END LOOP;
 25
 26      CLOSE get_sls;
 27
 28  END;
 29  /
Record is: Вася Пупкин Рапорт продажа 12.02.88
Record is: Максим Галкин Продано все 12.10.89

Процедура PL/SQL успешно завершена.
```

Вот так отработал наш с вами первый параметризованный курсор, надеюсь, все понятно, если нет - спрашивайте! Для закрепления, давайте запишем тот же курсор, с циклом **FOR**:

```
SET SERVEROUTPUT ON

DECLARE

  CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS
          SELECT * FROM SALESREPS
          WHERE AGE = INAGE;

BEGIN

  DBMS_OUTPUT.enable;

  FOR in_sls IN get_sls(37) LOOP

          DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||'
'||in_sls.TITLE||' '||in_sls.HIRE_DATE);

  END LOOP;

END;
/
```

Получаем:

```
SQL> DECLARE
  2
  3      CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS
  4              SELECT * FROM SALESREPS
  5              WHERE AGE = INAGE;
  6
  7  BEGIN
  8
  9      DBMS_OUTPUT.enable;
 10
 11      FOR in_sls IN get_sls(37) LOOP
 12
 13              DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||'
'||in_sls.TITLE||' '||in_sls.HIRE_DATE);
 14
 15      END LOOP;
 16
 17  END;
```

```
 18  /
Record is: Вася Пупкин Рапорт продажа 12.02.88
Record is: Максим Галкин Продано все 12.10.89

Процедура PL/SQL успешно завершена.
```

Параметров у курсора может быть несколько, например вот так:

```
SET SERVEROUTPUT ON

DECLARE

  CURSOR get_sls(INMG SALESREPS.MANAGER%TYPE, INSL SALESREPS.SALES%TYPE)
IS
          SELECT * FROM SALESREPS
          WHERE MANAGER = INMG AND SALES > INSL;

BEGIN

  DBMS_OUTPUT.enable;

  FOR in_sls IN get_sls(104, 300) LOOP

          DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||'
'||in_sls.TITLE||' '||TO_CHAR(in_sls.HIRE_DATE,'DD-MM-YYYY')||
          ' '||TO_CHAR(in_sls.SALES));

  END LOOP;

END;
/
```

Получаем после запуска:

```
SQL> DECLARE
  2
  3       CURSOR get_sls(INMG SALESREPS.MANAGER%TYPE, INSL
SALESREPS.SALES%TYPE) IS
  4               SELECT * FROM SALESREPS
  5               WHERE MANAGER = INMG AND SALES > INSL;
  6
  7  BEGIN
  8
  9       DBMS_OUTPUT.enable;
 10
 11       FOR in_sls IN get_sls(104, 300) LOOP
 12
 13               DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||'
'||in_sls.TITLE||' '||TO_CHAR(in_sls.HIRE_DATE,'DD-MM-YYYY')||
 14               ' '||TO_CHAR(in_sls.SALES));
 15
 16       END LOOP;
 17
 18  END;
 19  /
```

Иногда, при выборке из курсора бывает ситуация, что какой-либо столбец или строки результирующего набора необходимо обновить. То есть, изменить их содержимое. Для того, чтобы это осуществить, непосредственно при объявлении курсора необходимо использовать конструкцию - **FOR UPDATE** (для обновления ..). А, так же конструкцию, **WHERE CURRENT OF** (где текущая строка ..) в операторах **UPDATE**, **DELETE**. Собственно конструкция **FOR UPDATE**, является частью оператора **SELECT** и объявляется последней:

```
------ SELECT .... FROM .... FOR UPDATE [OF ссылка на столбец][NOWAIT]
--------
```

Где, собственно, "ссылка на столбец" это столбец таблицы, для которой выполнен запрос. Можно так же использовать список столбцов. Например, вот так:

```
DECLARE

  CURSOR get_sls IS
         SELECT * FROM SALESREPS
         FOR UPDATE OF SALESREPS.QUOTA, SALESREPS.SALES;
.
.
.
.

  -- Для столбцов QUOTA и SALES, таблицы SALESREPS.

DECLARE

  CURSOR get_sls(INMG SALESREPS.MANAGER%TYPE) IS
         SELECT * FROM SALESREPS
         WHERE MANAGER = INMG
         FOR UPDATE;
.
.
.
.

  -- Для всех столбцов таблицы SALESREPS.
```

Теперь немного теории. Так как обычный запрос с помощью оператора **SELECT**, при выполнении получает строки таблицы и при этом сама таблица выборки не блокируется, то есть любой другой пользователь может выполнить запрос к той же таблице, получив при этом данные. В **Oracle** при выполнении запроса, т.е. при извлечении активного набора **SELECT**, производится моментальный снимок таблицы (**snapshot**), при этом все изменения сделанные до этого момента кем-либо еще отражаются в данном наборе. А, после того как**snapshot** получен все изменения, произведенные в данной таблице выборке, даже если они зафиксированы оператором **COMMIT**, отражаться не будут!!! Для того, чтобы их отразить нужно закрыть и снова открыть курсор, загрузив данные заново! Это и есть алгоритм согласованного чтения данных, о котором я уже упоминал ранее. А вот когда мы объявляем **FOR UPDATE** - строки активного набора данных блокируются до момента выполнения **COMMIT**. Таким образом мы запрещаем изменение данных другим сеансам. Если какой-либо сеанс уже блокировал строки, то следующий **SELECT FOR UPDATE**, будет ждать снятия блокировки. В этом случае можно применить **NOWAIT** (без ожидания). Если обратиться к заблокированным строкам получим сообщение об ошибке **ORA-54**. Вот таким образом это работает. А вот конструкция **WHERE CURRENT OF** используется уже непосредственно при изменении данных:

```
------ WHERE CURRENT OF курсор ------
```

Где "курсор" - это курсор, открытый на обновление. Давайте рассмотрим практический пример такого курсора:

```
DECLARE

  CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
                        SELECT * FROM SALESREPS
                               WHERE MANAGER IN (
                                      SELECT O.MGR FROM OFFICES O
                                      WHERE TARGET > INTG)
                        FOR UPDATE OF SALESREPS.QUOTA;

BEGIN
```

```
  FOR get_cur_upd IN cur_upd(700) LOOP

  UPDATE SALESREPS
         SET SALESREPS.QUOTA = SALESREPS.QUOTA + 50
         WHERE CURRENT OF cur_upd;

  END LOOP;

COMMIT;

END;
/
```

После запуска в **SQL*Plus** получаем:

```
SQL> DECLARE
  2
  3   CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
  4      SELECT * FROM SALESREPS
  5       WHERE MANAGER IN (
  6         SELECT O.MGR FROM OFFICES O
  7         WHERE TARGET > INTG)
  8      FOR UPDATE OF SALESREPS.QUOTA;
  9
 10  BEGIN
 11
 12   FOR get_cur_upd IN cur_upd(700) LOOP
 13
 14   UPDATE SALESREPS
 15    SET SALESREPS.QUOTA = SALESREPS.QUOTA + 50
 16    WHERE CURRENT OF cur_upd;
 17
 18   END LOOP;
 19
 20  COMMIT;
 21
 22  END;
 23  /

Процедура PL/SQL успешно завершена.
```

Да, а что произошло? Просто строки столбца **QUOTA**, таблицы **SALESREPS**, соответствующие условию **TARGET > 700** увеличились, на 50! Что, можно проверить, выполнив простой запрос:

```
SELECT * FROM SALESREPS
/
```

Его посмотрите сами. Обратите внимание на то, что курсор выполнен с передачей параметра и использует цикл **LOOP**. Конструкция **FOR UPDATE OF SALESREPS.QUOTA** определяет обновляемый столбец, конструкция **WHERE CURRENT OF cur_upd** в операторе**UPDATE SALESREPS** определяет какие строки обновить. Кстати для закрепления, используя пакет **DBMS_OUTPUT** можете добавить код для того, чтобы было видно, что происходит. Оператор **COMMIT**, расположенный вне тела цикла, снимает блокировку с таблицы и фиксирует изменения. Давайте, с помощью другого блока, применив более компактный код, вернем все назад:

```
DECLARE

  CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
         SELECT * FROM SALESREPS
                WHERE MANAGER IN (
                        SELECT O.MGR FROM OFFICES O
                        WHERE TARGET > INTG)
                FOR UPDATE NOWAIT;
```

```
BEGIN

  FOR get_cur_upd IN cur_upd(700) LOOP

  UPDATE SALESREPS
         SET SALESREPS.QUOTA = SALESREPS.QUOTA - 50
         WHERE CURRENT OF cur_upd;

  END LOOP;

COMMIT;

END;
/
```

После запуска в **SQL*Plus** получаем:

```
SQL> DECLARE
  2
  3       CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
  4              SELECT * FROM SALESREPS
  5                     WHERE MANAGER IN (
  6                            SELECT O.MGR FROM OFFICES O
  7                            WHERE TARGET > INTG)
  8                     FOR UPDATE NOWAIT;
  9
 10  BEGIN
 11
 12       FOR get_cur_upd IN cur_upd(700) LOOP
 13
 14       UPDATE SALESREPS
 15              SET SALESREPS.QUOTA = SALESREPS.QUOTA - 50
 16              WHERE CURRENT OF cur_upd;
 17
 18       END LOOP;
 19
 20  COMMIT;
 21
 22  END;
 23  /

Процедура PL/SQL успешно завершена.
```

Здесь хорошо видно, что мы применили конструкцию **FOR UPDATE NOWAIT**, хотя в результате работы мы получили то, что нужно. Так же, применив курсорный цикл **FOR**, мы сделали более компактный код. Теперь, давайте побеседуем на тему оператора **COMMIT**. Посмотрим вот такой блок:

```
DECLARE

  CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
         SELECT * FROM SALESREPS
                WHERE MANAGER IN (
                       SELECT O.MGR FROM OFFICES O
                       WHERE TARGET > INTG)
                FOR UPDATE NOWAIT;

  get_cur_upd  cur_upd%ROWTYPE;

BEGIN

  OPEN cur_upd(700);
```

```
  FETCH cur_upd INTO get_cur_upd;

  COMMIT WORK;

  FETCH cur_upd INTO get_cur_upd;

END;
/
```

Получаем:

```
SQL> DECLARE
  2
  3      CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
  4              SELECT * FROM SALESREPS
  5                      WHERE MANAGER IN (
  6                              SELECT O.MGR FROM OFFICES O
  7                              WHERE TARGET > INTG)
  8                      FOR UPDATE NOWAIT;
  9
 10      get_cur_upd  cur_upd%ROWTYPE;
 11
 12  BEGIN
 13
 14      OPEN cur_upd(700);
 15
 16      FETCH cur_upd INTO get_cur_upd;
 17
 18      COMMIT WORK;
 19
 20      FETCH cur_upd INTO get_cur_upd;
 21
 22  END;
 23  /
DECLARE
*
ошибка в строке 1:
ORA-01002: выборка из последовательности
ORA-06512: на  line 20
```

Оператор **COMMIT WORK** снял блокировку с таблицы и последующая выборка привела к ошибке. Следовательно, как в предыдущих примерах, располагать **COMMIT** нужно после цикла. А что, если нужно обновить строки из курсора не применяя конструкции **FOR UPDATE** ? Можно, если у таблицы есть уникальный ключ, вот так:

```
DECLARE

  CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
         SELECT * FROM SALESREPS
                 WHERE MANAGER IN (
                         SELECT O.MGR FROM OFFICES O
                         WHERE TARGET > INTG);

BEGIN

  FOR get_cur_upd IN cur_upd(700) LOOP

  UPDATE SALESREPS
         SET SALESREPS.QUOTA = SALESREPS.QUOTA - 10
         WHERE EMPL_NUM = get_cur_upd.EMPL_NUM;
```

```
    -- Is not FOR UPDATE --
    COMMIT WORK;

    END LOOP;

    COMMIT;

  END;
  /
```

Получаем:

```
SQL> DECLARE
  2
  3       CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
  4               SELECT * FROM SALESREPS
  5                       WHERE MANAGER IN (
  6                               SELECT O.MGR FROM OFFICES O
  7                               WHERE TARGET > INTG);
  8
  9
 10  BEGIN
 11
 12       FOR get_cur_upd IN cur_upd(700) LOOP
 13
 14       UPDATE SALESREPS
 15               SET SALESREPS.QUOTA = SALESREPS.QUOTA - 10
 16               WHERE EMPL_NUM = get_cur_upd.EMPL_NUM;
 17
 18       -- Is not FOR UPDATE --
 19       COMMIT WORK;
 20
 21       END LOOP;
 22
 23       COMMIT;
 24
 25  END;
 26  /
```

Задание:

Напишите код PL/SQL, который бы выводил информацию из таблицы hr.employees в виде, аналогичном представленному на рис. Лаб. 17.1-1. Должны выводиться:

- информация о номере сотрудника;
- информация о фамилии и имени сотрудника;
- информация о времени, которое этот сотрудник отработал на предприятии. Если время более и равно 10 лет, должен выводиться текст "Больше или равно 10 лет". В противном случае должен выводиться текст "Меньше 10 лет".

Используйте для этого кода явный курсор.

```
Номер сотрудника: 198 Имя и фамилия: Donald OConnell Меньше 10 лет
Номер сотрудника: 199 Имя и фамилия: Douglas Grant Меньше 10 лет
Номер сотрудника: 200 Имя и фамилия: Jennifer Whalen Больше или равно 10 лет
Номер сотрудника: 201 Имя и фамилия: Michael Hartstein Больше или равно 10 лет
Номер сотрудника: 202 Имя и фамилия: Pat Fay Больше или равно 10 лет
Номер сотрудника: 203 Имя и фамилия: Susan Mavris Больше или равно 10 лет
Номер сотрудника: 204 Имя и фамилия: Hermann Baer Больше или равно 10 лет
```

Рис. Лаб. 17.1-1 (показаны частичные результаты).

Решение:

Соответствующий код PL/SQL может быть таким (выполняется в SQL*Plus):

```
DECLARE

rEmp hr.Employees%ROWTYPE;

CURSOR cCur1 IS SELECT * FROM hr.employees;

BEGIN

Open cCur1;

LOOP

FETCH cCur1 into rEmp;

EXIT WHEN cCur1%NOTFOUND;

If MONTHS_BETWEEN(SYSDATE, rEmp.Hire_date)/12 >= 10 then

DBMS_Output.PUT_LINE('Номер сотрудника:  '  ||  rEmp.Employee_ID  ||
' Имя и фамилия: ' || rEmp.first_name || ' ' || rEmp.last_name ||
' Больше или равно 10 лет');

ELSE

DBMS_Output.PUT_LINE('Номер сотрудника:  '  ||  rEmp.Employee_ID  ||
' Имя и фамилия: ' || rEmp.first_name || ' ' || rEmp.last_name ||
' Меньше 10 лет');

END IF;

END LOOP;

CLOSE cCur1;

END;
```