Pro*C 사용 방법 안내(Ver 0.9)

이 문서는 오라클의 Pro*C/C++ Precompiler Programmer's Guide.pdf 문서의 일부분을 정리한 것입니다. 보다 자세한 사용법은 위 문서를 참고하시길 바랍니다.

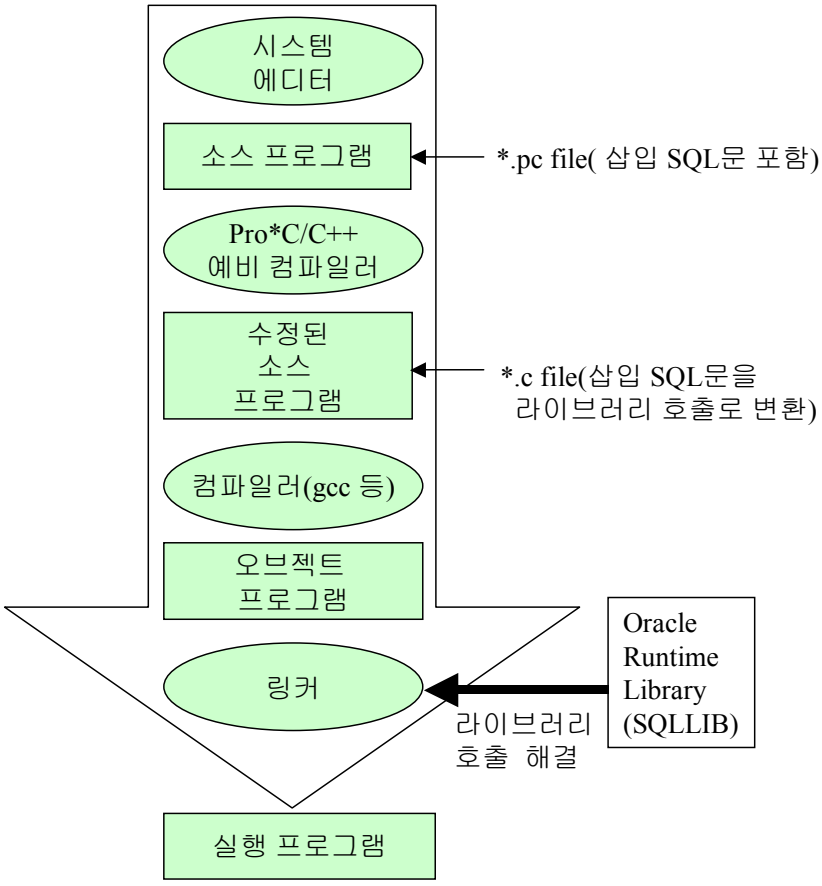
I. 개요 3
II. 환경 설정 4
III. Pro*C/C++ 프로그램 가이드 5
3.1 용어 설명 5
3.2 삽입 SQL 문장의 사용 5
3.3 변수 선언 5
3.4 선언부(declare section) 6
3.5 SQL 문의 사용법 6
3.5.1 SELECT 문 6
3.5.2 DML 문 7
3.5.3 커서의 조작 8
3.5.4 VARCHAR 변수 9
3.5.5 호스트 구조체(host structure) 11
3.5.6 호스트 배열(host array) 12
3.5.7 지시 변수(indicator variable) 14
3.5.8 INCLUDE 문 15
3.6 에러 처리 16
3.6.1 WHENEVER문 16
3.6.2 sqlca의 사용 19
3.6.3 oraca를 이용하는 방법 22
3.7 오라클 접속 및 해제 24
3.7.1 접속 24
3.7.2 원격에서 접속하는 경우 25
3.7.3 접속 종료 25
3.8 컴파일 및 실행 26
IV. 예제 설명 27
4.1 Pro*C 예제 27

I. 개요

오라클의 예비 컴파일러(precompiler)는 응용 프로그램 안에 직접 SQL문을 사용할 수 있게 해주는 도구이다. 예비 컴파일러는 소스 프로그램을 받아 들여 그 속에 삽입(embed)된 SQL 문장을 표준 Oracle runtime library 호출로 바꾸어 수정된 프로그램 소스 코드를 생성한다. 사용자는 이 소스 코드를 일반적인 방법으로 컴파일하고, 링크하고, 수행한다.

오라클은 C와 C++을 위해 Pro*C/C++ 이라는 예비 컴파일러를 제공하며 일반적으로 삽입 SQL문이 포함된 소스 파일의 확장자는 ".pc"이다. 이 pc 파일을 예비 컴파일러를 통하여 컴파일 하면 C파일이 생성된다.

Pro*C를 이용한 응용 프로그램의 개발 절차는 아래 그림과 같다.



II. 환경 설정

Redhat 리눅스를 기반으로 설명한다.
pcscfg.cfg 파일은 ProC를 사용하기 위한 시스템 설정 파일이다. $ORACLE_HOME/precomp/admin 에 pcscfg.cfg 파일이 있다. 이 파일에 다음과 같은 내용을 추가한다.

```
sys_include=($ORACLE_HOME/precomp/public, /usr/include, /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include)
include=($ORACLE_HOME/precomp/public)
include=($ORACLE_HOME/rdbms/demo)
include=($ORACLE_HOME/network/public)
include=($ORACLE_HOME/plsql/public)
ltype=short
```

sys_include 부분이 두 행에 걸쳐 있는데, 실제 사용할 때는 한 라인으로 작성해야 한다. /usr/lib/gcc-lib/i386-redhat-linx/egcs-2.91.66/include 부분은 각자의 리눅스 환경에 맞도록 변경해야 한다. /usr/lib/gcc-lib/i386-redhat-linux 디렉토리는 대부분 존재할 것이다. 그 다음 하위 디렉토리를 여러분의 환경에 맞도록 수정한다.

III. Pro*C/C++ 프로그램 가이드

3.1 용어 설명

- 호스트 프로그램 : 삽입 SQL 문이 포함된 프로그램
- 호스트 언어 : 삽입 SQL을 포함한 C, C++, JAVA, COBOL 등을 지칭
- 호스트 변수 : 삽인 SQL문에서 조건에 사용되는 값을 전달하거나 검색한 값들을 프로그램에 전달하는 변수

3.2 삽입 SQL 문장의 사용

응용 프로그램 상에서, 완전한 SQL문장과 완전한 C 문장을 혼합해서 사용할 수 있고, C의 변수와 구조체(structure)를 SQL 문에서 사용할 수 있다.
호스트 프로그램에서 SQL 문장을 사용하려면, EXEC SQL이라는 키워드를 사용하여 시작하고 세미콜론으로 종료한다.
C의 변수들을 SQL 문에서 사용하려면 다른 SQL 필드 이름과 구별하기 위해 콜론(:)을 앞에 붙여 사용한다.

```
…
EXEC SQL SELECT empno, ename, job
INTO :empno, :ename, :job
FROM emp
WHERE deptno = :deptno ;
…
```

위의 SQL 문에서 SELECT 다음의 empno, ename, job 은 emp 테이블의 필드 이름을 나타내며, INTO 다음의 empno, ename, job은 호스트 변수를 의미한다. 즉 위의 문장은 emp 테이블에서 주어진 부서 번호(deptno)에 속하는 직원들의 번호(empno), 이름(ename), 직업(job)을 가져와서 호스트 변수 empno, ename, job에 저장하는 것이다.

3.3 변수 선언

호스트 변수는 아래의 표1과 같은 타입으로 선언할 수 있다.

데이타 타입	설명

Char	한 문자
Char[n]	N개의 문자 배열(문자열)
int	정수
short	작은 정수
long	큰 정수
float	부동 소수점수(단정도형)
double	부동 소수점수(배정도형)
VARCHAR[n]	가변 길이 문자열

3.4 선언부(declare section)

호스트 변수를 C 언어의 규칙에 따라 선언부에서 선언해야 한다. 예비 컴파일 옵션중 MODE=ORACLE 이면, 특별한 선언부에 선언할 필요가 없다. CODE=CPP 옵션이면(C++을 사용한다면), 반드시 선언부가 있어야 한다.

```
EXEC SQL BEGIN DECLARE SECTION;
    /* 모든 호스트 변수를 선언 */
    char *uid = "scott/tiger";
    ...
EXEC SQL END DECLARE SECTION;
```

선언부에는 아래와 같은 것들을 포함할 수 있다.

- 호스트 변수
- 호스트 변수가 아닌 C/C++ 변수
- EXEC SQL INCLUDE 문
- EXEC SQL ORACLE 문
- C/C++ 주석

3.5 SQL 문의 사용법

3.5.1 SELECT 문

SELECT 문을 실행할 때는 SELECT 문이 반환하는 데이타 행의 수를 다루어야 한다. SELECT 문장은 아래와 같이 구분할 수 있다.

- 어떤 행도 반환하지 않는 질의
- 한 행만 반환하는 질의
- 한 행 이상을 반환하는 질의

한 행 이상을 반환하는 질의는 명시적으로 선언된 커서(cursor)나 호스트 배열의 사용을 필요로 한다.
여기서는 우선 하나의 행만을 반환하는 SELECT 문에 대해서 언급한다. 커서의 사용법은 아래에서 구체적으로 다룬다.

SELECT 문의 예는 아래와 같다.

```
EXEC SQL SELECT ename, job, sal + 2000
    INTO :emp_name, :job_title, :salary
    FROM emp
    WHERE empno = :emp_number;
```

오라클은 INTO 절에 있는 호스트 변수들에 반환 SELECT 문의 반환 값을 저장한다. 주의할 것은 SELECT 절에 있는 컬럼의 수와 INTO 절에 있는 호스트 변수의 수가 같아야 한다.

3.5.2 DML 문

INSERT, UPDATE, DELETE 문을 사용하는 방법은 아래와 같다. INSERT의 경우 VALUE 절에 적절한 호스트 변수를 전달한다. UPDATE는 WHERE 절과 SET절에 호스트 변수를 명세하여 사용할 수 있고, DELETE는 WHERE 절에 호스트 변수를 명세하면 된다.

```
...
EXEC SQL INSERT INTO emp (empno, ename, sal, deptno)
    VALUES (:emp_number, :emp_name, :salary, :dept_number);

EXEC SQL UPDATE emp
    SET sal = :salary, comm = :commission
    WHERE empno = :emp_number;

EXEC SQL DELETE FROM emp
    WHERE deptno = :dept_number ;
...
```

3.5.3 커서의 조작

커서를 다루기 위해서는 아래의 명령이 필요하다.

삽입 SQL 문	설명
DECLARE	커서의 이름을 명명하고, SELECT 문과 연관 시킴
OPEN	질의를 수행하고, 결과 집합을 명시
FETCH	결과 집합에서 하나의 행을 가져오고, 커서를 이동시킴
CLOSE	커서를 닫음

아래의 예와 같이 DECLARE CURSOR문을 이용하여 emp_cursor라는 이름의 커서를 정의한다. 커서와 연관된 SELECT 문에 INTO 절을 포함할 수 없다. 커서의 경우 FETCH 명령에서 INTO 절을 사용한다.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, empno, sal
    FROM emp
    WHERE deptno = :dept_number;
```

커서의 선언은 다른 커서 조작 명령(OPEN, FETCH, CLOSE)보다 앞에 위치해야 한다. 커서 조작 명령은 하나의 단위로 컴파일 되어야 한다. 예를 들어 파일 A에서 DECLARE를 하고 파일 B에서 OPEN할 수 없다.

아래의 예와 같이 emp_cursor를 오픈할 수 있다.

```
EXEC SQL OPEN emp_cursor;
```

OPEN 명령은 결과 집합의 첫번째 행의 바로 앞을 지정하고 있는 상태이다.

FETCH 명령을 이용하여 결과 집합에서 행들을 가져오고, 호스트 변수의 결과들을 저장하도록 한다. FETCH 명령은 아래와 같이 사용할 수 있다.

```
EXEC SQL FETCH emp_cursor
    INTO :emp_name, :emp_number, :salary;
```

여러 결과를 다 가져오기 위해서, 보통 FETCH 명령은 아래 처럼 루프 속에서 사용된다.

```
EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;)
```

```
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :emp_number, :salary;
    printf("Name : %s", emp_name);    /* 이름 출력 */
    …
}
```

결과 집합이 비었거나 더 이상의 행을 가지고 있지 않다면, FETCH 명령은 "no data found"에러를 발생한다. 일반적으로 위의 예처럼 무한 루프를 빠져 나오기 위해 WHENEVER NOT FOUND를 사용하게 된다. WHENEVER 문은 에러 처리에서 자세히 설명한다.

결과들을 다 가져왔으면, 자원을 해제하기 위해 커서를 종료한다.

```
EXEC SQL CLOSE emp_cursor;
```

```
int emp_number;
char temp[20];
VARCHAR emp_name[20];

/* 호스트 변수에 값 지정*/
printf("Employee number? ");
gets(temp);
emp_number = atoi(temp);
printf("Employee name? ");
gets(emp_name.arr);
emp_name.len = strlen(emp_name.arr);

EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
        VALUES (:emp_number, :emp_name);
```

3.5.4 VARCHAR 변수

가변 길이 문자열을 선언하기 위해 VARCHAR 슈도 타입을 사용한다. VARCHAR2 컬럼의 값을 처리할 때, C의 표준 문자열 보다 VARCHAR를 사용하는 것이 더 편리하다.
VARCHAR를 구조체로 생각하면 된다. 예를 들어 아래와 같이 VARCHAR로 선언하면 예비

컴파일러는 arr과 len을 멤버로 갖는 구조체로 확장한다.

```
VARCHAR username[20];

struct {
        unsigned short len;
        unsigned char arr[20];
} username;
```

VARCHAR를 이용하면, SELECT나 FETCH 이후에 명시적으로 VARCHAR 구조체의 len 필드를 사용할 수 있다.
따라서 아래와 같은 코드를 사용하여 arr 필드에 널(null) 문자를 추가하거나, 길이 필드를 이용하여 strncpy나 printf 함수를 사용한다.

```
username.arr[username.len] = '₩0';
…
printf("Username is %.*s₩n", username.len, username.arr);
```

SQL 문에서는, 구조체 이름에 콜론(:)을 붙여서 VARCHAR 변수를 참조한다.

```
…
int part_number;
VARCHAR part_desc[40];
…
main(){
        …
        EXEC SQL SELECT pdesc INTO :part_desc
                FROM parts
                WHERE pnum = :part_number;
        …
```

질의가 실행된 이후에, part_desc.len은 part_desc.arr에 저장된 문자열의 실제 길이를 저장한다.
C 문장에서는, 구조체의 접근 방식으로 각각의 필드들을 이용한다.

```
printf("₩n₩nEnter part description: ");
```

```
gets(part_desc.arr);
/* INSERT나 UPDATE에서 VARCHAR를 사용하기 전에 길이를 설정해야 한다. */
part_desc.len = strlen(part_desc.arr);
```

3.5.5 호스트 구조체(host structure)

호스트 변수들을 저장하기 위해 C의 구조체를 사용할 수 있다. SELECT나 FETCH 문의 INTO절, INSERT 문의 VALUE 절에서 호스트 변수를 저장하고 있는 구조체를 참조할 수 있다.

구조체가 호스트 변수로 사용될 때, SQL 문에서는 구조체의 이름만 사용되지만 각각의 필드들은 ORACLE에 데이타를 보내거나 ORACLE로부터 데이타를 받게 된다.

아래는 emp 테이블에 호스트 구조체를 사용하여 데이타를 추가하는 예제이다.

```
typedef struct {
        char emp_name[11];    /*  컬럼 길이보다 하나 크게 선언 */
        int emp_number;
        int dept_number;
        float salary;
} emp_record;
...
/* emp_record 타입의 변수 선언 */
emp_record new_employee;
strcpy(new_employee.emp_name, "CHEN");
new_employee.emp_number = 9876;
new_employee.dept_number = 20;
new_employee.salary = 4250.00;

EXEC SQL INSERT INTO emp (ename, empno, deptno, sal)
        VALUES (:new_employee );
```

구조체에 선언된 필드들의 순서와 SQL 문에서 관련된 컬럼들의 순서가 일치해야 된다. 아래의 예는 순서가 맞지 않아서 에러를 발생하는 예이다.

```
struct {
        int empno;
        float salary;
```

```
        char emp_name[10];
} emp_record;
...
/* 구조체 필드의 순서와 대응되는 컬럼 순서의 불일치 */
SELECT empno, ename, sal
        INTO :emp_record FROM emp;
```

3.5.6 호스트 배열(host array)

배열을 사용하면, 하나의 SQL 문으로 배열의 전체 원소를 조작할 수 있다. 따라서 특히 네트워크 환경에서, ORACLE 통신 오버헤드가 현저하게 줄어든다. 예를 들어, 300명의 직원에 관한 정보를 EMP 테이블에 추가하길 원한다고 하자. 배열이 없다면 300개의 INSERT 문을 실행해야 한다. 배열을 사용한다면, 하나의 INSERT 문으로 가능하다.

배열은 다음과 같이 선언한다.

```
char emp_name[50][10];
int emp_number[50];
float salary[50];
VARCHAR v_array[10][30];
```

문자 배열(스트링)을 제외하고는, SQL 문에서 참조할 수 있는 배열은 1차원으로 제한되어 있이다. .

```
int hi_lo_scores[25][25];    /* 허용되지 않음 */
VARCHAR name[2][20];    /* 허용됨 */
char     phone[2][14];    /* 허용됨 */
```

SELECT 문의 INTO 절에 호스트 배열을 사용할 수 있다. SELECT 가 반환하는 행의 최대 수를 안다면, 최대 수 만큼의 호스트 배열을 선언하면 된다. 아래의 예와 같이 사용할 수 있다.

```
char emp_name[50][20];
int emp_number[50];
float salary[50];

EXEC SQL SELECT ENAME, EMPNO, SAL
        INTO :emp_name, :emp_number, :salary
        FROM EMP
```

```
        WHERE SAL > 1000;
```

위의 예에서, SELECT 문은 최대 50개의 행만 반환한다. 만약 50개의 행 이상을 반환한다면, 위의 방법으로 모든 행을 가져올 수 없다. 더 큰 배열을 사용하거나 커서를 선언한 후 FETCH 문을 사용해야 한다.

호스트 배열을 INSERT 문에서 사용하는 예는 아래와 같다.

```
char emp_name[50][20];
int emp_number[50];
float salary[50];
/* 호스트 배열에 데이타 지정 */
...
EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
          VALUES (:emp_name, :emp_number, :salary);
```

UPDATE 문에도 호스트 배열을 사용할 수 있다.

```
int emp_number[50];
float salary[50];

/* 호스트 배열에 데이타 지정 */
EXEC SQL UPDATE emp SET sal = :salary
WHERE EMPNO = :emp_number;
```

호스트 배열을 호스트 구조체의 필드로 사용할 수 있다.

```
struct {
        char emp_name[3][10];
        int emp_number[3];
        int dept_number[3];
} emp_rec;
...
strcpy(emp_rec.emp_name[0], "ANQUETIL");
strcpy(emp_rec.emp_name[1], "MERCKX");
strcpy(emp_rec.emp_name[2], "HINAULT");
emp_rec.emp_number[0] = 1964; emp_rec.dept_number[0] = 5;
emp_rec.emp_number[1] = 1974; emp_rec.dept_number[1] = 5;
emp_rec.emp_number[2] = 1985; emp_rec.dept_number[2] = 5;
```

```
EXEC SQL INSERT INTO emp (ename, empno, deptno)
        VALUES ( :emp_rec);
...
```

3.5.7 지시 변수(indicator variable)

모든 호스트 변수에 부가적인 지시 변수를 관련시킬 수 있다. 일반적으로 지시 변수는 입력으로 사용되는 호스트 변수에 NULL을 할당하고, 출력으로 사용되는 호스트 변수에 NULL이 오거나 잘려진 값이 저장되는지 검사하기 위해 사용된다.
모든 지시 변수는 2 바이트 숫자형(short)로 선언되어야 한다. SQL에서 사용법은 아래와 같다.

```
:host_variable INDICATOR :indicator_variable
또는
:host_variable:indicator_variable
```

지시 변수 값의 의미는 아래 표와 같다.

값	의미
0	연산이 성공적으로 수행됨
-1	NULL이 반환되거나, 삽입되거나, 변경됨
-2	호스트 변수에 잘려진 값이 저장되고, 컬럼 값의 원래 길이도 알 수 없음
> 0	호스트 변수에 잘려진 값이 저장되고, 반환된 값은 컬럼 값의 원래 길이를 의미

아래의 예와 같이 지시 변수 SELECT 문에서 사용할 수 있다.

```
EXEC SQL BEGIN DECLARE SECTION;
        int emp_number;
        float salary, commission;
        short comm_ind;  /* 지시 변수 */
EXEC SQL END DECLARE SECTION;
        char temp[16];
        float pay; /* SQL 문에서 사용되지 않는 변수들 */
...
printf("Employee number? ");
gets(temp);
emp_number = atof(temp);
EXEC SQL SELECT SAL, COMM
        INTO :salary, :commission:ind_comm
```

```
        FROM EMP
        WHERE EMPNO = :emp_number;
if(ind_comm == -1)    /* commission 이 NULL인지 확인 */
        pay = salary;
else
        pay = salary +  commission;
```

INSERT에 NULL을 주기 위해서 아래와 같이 사용할 수 있다. 아래의 예는 사용자의 입력에 따라 emp_number의 값을 유연하게 입력하는 것을 보여준다.

```
printf("Enter employee number or 0 if not available: ");
scanf("%d", &emp_number);
if (emp_number == 0)
        ind_empnum = -1;
else
        ind_empnum = 0;
EXEC SQL INSERT INTO emp (empno, sal)
        VALUES ( :emp_number:ind_empnum, :salary);
```

3.5.8 INCLUDE 문

삽입 SQL을 이용하여 응용 프로그램을 작성하기 위해서는 sqlca.h나 oraca.h 파일을 포함해야 한다. 아래와 같은 방법으로 포함할 수 있다.

```
#include <sqlca.h>
#include <oraca.h>
또는
EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE oraca;
```

여기서 sqlca는 SQL Communication Area를 뜻하며 내부적으로는 sqlca라는 구조체가 선언된 화일이다. 데이타베이스는 SQL문의 실행이 끝날 때마다 sqlca 구조체에 알맞은 값을 지정하여 호스트 프로그램에서 참조할 수 있게 해, 데이타베이스와 호스트 프로그램 사이의 통신 역할을 하게 한다. sqlca에 담겨지는 정보는 경고 메시지, 에러 발생 여부 및 에러 코드 등이다. 자세한 내용은 다음 절에서 설명하고 있다.

oraca(Oracle Communication Area)는 sqlca라는 SQL 표준에 덧붙여서 오라클에서 확장한 구조체로서 sqlca에서 제공되는 정보 외에 추가로 필요한 정보를 호스트 프로그램에게 제공하기 위한 구조체이다.

이 oraca를 사용하기 위해서는 다음과 같은 문장을 통해 ORACA=YES라는 값을 지정하여

oraca를 활성화시켜야 한다.

```
EXEC ORACLE OPTION (ORACA=YES);
```

그러나, ORACA를 사용하면 프로그램 실행시 추가 작업으로 인한 부담이 늘어나므로 보통의 경우에는 위의 문장을 생략해서 기본값인 ORACA=NO로 지정하여 oraca를 활성화시키지 않는다.

3.6 에러 처리

Pro*C/C++을 이용한 응용 프로그램 작성에서 SQL문의 실행에 따른 에러 처리 및 진단의 방법은 크게 세 가지로 나눌 수 있다.

- WHENEVER문을 이용하는 방법
- sqlca를 이용하는 방법
- oraca를 이용하는 방법

3.6.1 WHENEVER문

WHENEVER문의 구문은 다음과 같다.

```
EXEC SQL WHENEVER <condition> <action>;
```

<condition>에는 아래의 표와 같은 것을 사용할 수 있다.

조건	의미
SQLWARNING	경고 메시지가 발생한 경우
SQLERROR	에러가 발생한 경우
NOT FOUND	WHERE절의 조건을 만족하는 행을 발견할 수 없거나, SELECT INTO나 FETCH가 어떤 행도 반환하지 않은 경우

<action>에는 아래의 표와 같은 것을 지정할 수 있다.

조치	의미
CONTINUE	프로그램은 다음 문장을 수행함. 디폴트 동작으로 WHENEVER를 쓰지 않은 경우와 같음
DO	에러 처리 함수를 수행. 에러 처리 함수 수행 후 실패한 SQL문의 다음 문장으로 제어가 넘어감
DO BREAK	실제적인 “break” 문장. 루프에서 사용

DO CONTINUE	실제적인 “continue” 문장. 루프에서 사용
GOTO label_name	레이블로 제어 이동.
STOP	프로그램 수행을 종료하고, 트랜잭션은 롤백됨

WHENEVER문이 정의되면 그 이후의 SQL문을 실행할 때마다 <조건>에 해당되는지를 검사하고 해당되면 사건이 일어날 때마다 <조치>에 정의된 동작을 취하게 된다. 이 WHENEVER문은 다른 조건이나 조치로 구성된 WHENEVER문을 만날 때까지 계속 유효하게 동작한다.

WHENEVER문의 사용 예를 보자.

```
main()
{
        .....
        EXEC SQL DECLARE emp_cursor CURSOR FOR   /* 커서 설정 */
                SELECT empno, ename FROM emp;
        EXEC SQL OPEN emp_cursor;   /* 커서를 연다 */

        EXEC SQL WHENEVER NOT FOUND DO break;
        while(1)
        {
                EXEC SQL FETCH emp_cursor INTO :emp_number, :emp_name;
                .......
        }
        EXEC SQL CLOSE emp_cursor; /* 커서를 닫는다 */
          ....
}
```

위 예의 while루프에서 FETCH문이 실행될 때 더 이상 레코드가 발견되지 않으면 WHENEVER문의 조치에 의해 루프를 빠져 나오게 된다.

```
main()
{
        …
        EXEC SQL WHENEVER SQLERROR DO sql_error("에러 메시지");

        EXEC SQL SELECT ....
```

```
        …
        EXEC SQL UPDATE .....
        …
}

sql_error(char * errmasage)
{
          printf("에러 %s₩n", errmasage);
          exit(-1);
}
```

위 예에서는 WHENEVER문이 정의된 이후에 실행되는 SQL문에서 에러가 발생하면 sql_error()라는 함수를 실행하게 된다. 즉 SELECT가 실패해도 sql_error()가 호출되고, UPDATE가 실패해도 sql_error()가 호출된다.

또 다음과 같은 예를 보자.

```
        /* 무한 루프에 빠지는 예 */
        …
        EXEC SQL WHENEVER NOT FOUND GOTO no_more;
        while(1)
        {
                EXEC SQL FETCH emp_cursor INTO :emp_number, :emp_name;
                …
        }

no_more:
          EXEC SQL DELETE FROM emp WHERE empno=:emp_number;
          ....
}
```

위의 예를 보면 FETCH에서 더 이상 해당되는 행이 없는 경우 no_more로 넘어가게 되고 no_more 다음의 DELETE문에서 다시 NOT FOUND조건에 해당되는 경우 다시 no_more로 넘어가서 무한 루프에 빠지게 되는 것을 알 수 있다.

아래의 예와 같이 GOTO를 다시 설정하여 무한 루프를 방지한다.

```
        …
        EXEC SQL WHENEVER NOT FOUND GOTO no_more;
```

```
        for (;;)
        {
                EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
                ...
        }
no_more:
        EXEC SQL WHENEVER NOT FOUND GOTO no_match;
        EXEC SQL DELETE FROM emp WHERE empno = :emp_number;
        ...
no_match:
        ...
```

3.6.2 sqlca의 사용

SQLCA는 구조체이다. SQLCA의 필드는 에러, 경고, SQL이 수행될 때 마다 오라클이 갱신하는 상태 정보를 담고 있다. 따라서 SQLCA는 가장 최근의 SQL 연산의 결과를 반영한다. SQLCA를 사용하기 위해서는 아래와 같은 문장을 포함해야 한다.

```
EXEC SQL INCLUDE SQLCA;
또는
#include <sqlca.h>
```

sqlca 구조체의 구조는 다음과 같다.

```
struct sqlca
{
  char   sqlcaid[8];              /* "SQLCA" 문자 스트링 */
  long   sqlabc;                  /* slqca구조체의 길이 */
  long   sqlcode;                 /* 에러코드 */
  struct
  {
    unsigned short sqlerrml;      /* 에러 메시지 길이 */
    char           sqlerrmc[70];  /* 에러 메시지 내용 */
  } sqlerrm;
  char   sqlerrp[8];              /* reserved */
  long   sqlerrd[6];              /* sqlerrp[0] : reserved
                                     sqlerrp[1] : reserved
```

```
                                sqlerrd[2] : 수행된 행의 개수
                                sqlerrd[3] : reserver
                                sqlerrd[4] : 파싱 에러 옵셋
                                sqlerrd[5] : reserved */
  char   sqlwarn[8];          /* sqlwarn[0] : 경고 플래그
                                sqlwarn[1] : 문자스트링이 절단된 경우
                                sqlwarn[2] : 안쓰임.
                                sqlwarn[3] : SELECT문에서 필드 개수와
                                             INTO문의 호스트 변수 개수가
                                            일치하지 않음
                                sqlwarn[4] : DELETE 또는 UPDATE문에서
                                             where절이 없음.
                                sqlwarn[5] : reserved
                                sqlwarn[6] : 안쓰임.
                                sqlwarn[7] : 안쓰임. */
  char   sqlext[8];           /* reserved */
  };
```

sqlca.sqlcode는 SQL문 실행시 발생한 에러 코드이며 그 값에 따라 다음과 같은 뜻이 있다.

Sqlcode 값	의미
0	성공
양수	SQL문의 실행에는 성공했지만 예외 상황이 발생한 경우이다. SELECT INTO 또는 FETCH문에서 WHERE절의 조건에 만족하는 행이 없는 경우에 양수의 sqlcode가 반환된다.
음수	SQL문의 실행에 실패한 경우이다. 따라서 대부분의 경우 트랜잭션을 복귀(rollback)하여야 한다.

sqlca.sqlcode를 통해 알아낸 에러 코드의 의미와 원인 및 조치를 알고 싶으면 prompt상에서 oerr라는 명령을 사용한다. oerr 명령은 두 개의 인수를 사용한다. 첫 번째는 에러 메시지를 발생시킨 요소의 코드(예를 들어 ora는 오라클 서버)이며 두 번째 인수는 에러 코드 번호이다.

prompt> oerr ora 에러 코드의 절대값

여기서 에러코드는 절대값을 입력하여야 한다. 즉, 다음과 같이 -942 에러가 발생하였다면 942를 입력하여야 한다.

prompt> oerr ora 942

sqlwarn[0]부터 sqlwarn[7]까지는 경고에 관한 요소이며 이들을 참조함으로써 보다 상세한 제어를 할 수 있다.

sqlca.sqlerrd[2]는 SQL문의 영향을 받은 행의 개수를 담고 있다. 즉, SELECT, DELETE, UPDATE문 등의 실행 결과가 반영된 행의 개수를 말하며 sqlca.sqlcode와 더불어 가장 많이 참조되는 요소이다.
sqlca.sqlerrm.sqlerrmc는 에러 메시지를 담고 있으며 이를 참조하여 다음과 같이 에러 메시지를 출력할 수 있다.

```
sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml]='₩0';
printf("%s", sqlca.sqlerrm.sqlerrmc);
```

sqlerrmc에는 최대 70자까지의 에러 메시지만 저장되므로, 에러 메시지 전체를 알고 싶으면 sqlglm()을 사용할 수 있다.
sqlglm()의 구문은 다음과 같다.

```
sqlglm(message_buffer, &buffer_size, &message_length);
```

여기서 각 매개변수의 뜻은 다음과 같다.

- message_buffer : 에러 메시지가 담겨질 문자 배열.
- buffer_size : message_buffer의 크기.
- message_length : message_buffer에 담겨진 메시지의 크기.

예를 보자.

```
main()
{
        char msg_buffer[100];
        int     buf_size = 100;
        int     msg_length;
```

```
        …
        if (sqlca.sqlcode!=0) {
                sqlglm( msg_buffer, &buf_size, &msg_length);
                msg_buffer[msg_length] = '₩0';
                printf("%s", msg_buf);
        }
        …
}
```

3.6.3 oraca를 이용하는 방법

SQLCA가 제공하는 런타임 에러와 상태 변화와 같은 정보를 더 자세히 알고 싶을 때, ORACA를 사용한다. ORACA는 확장된 진단 툴을 제공한다. 그러나 ORACA는 런타임 오버헤드를 추가하므로, ORACA를 선택적으로 사용한다.

ORACA를 선언하기 위해서는 아래와 같은 문장을 프로그램에 추가해야 한다.

```
EXEC SQL INCLUDE ORACA;
또는
#include <oraca.h>
```

ORACA를 활성화하려면, ORACA 옵션을 명세해야 한다.

```
EXEC ORACLE OPTION (ORACA=YES);
```

ORACA 구조체의 정보는 다음과 같다.

```
struct oraca
{
  char   oracaid[8];              /* "ORACA" 문자 스트링 */
  long   oracabc;                 /* oraca의 크기 */
  long   oracchf;                 /* 커서 캐쉬 일관성 플래그 */
  long   oradbgf;                 /* 마스터 디버그 플래그 */
  long   orahchf;                 /* 히프 일관성 플래그 */
  long   orastxtf;                /* Save-SQL-statement 플래그 */
  struct
  {
    unsigned short orastxtl;      /* 현재 SQL문의 길이 */
    char           orastxtc[70];  /* 현재 SQL문의 내용 */
```

```
  } orastxt;
  struct
  {
    char    orasfnmc[70];        /* 현재 SQL문을 담고 있는 화일이름 */
    unsigned short orasfnml;     /* 화일이름의 길이 */
  } orasfnm;
  long   oraslnr;                /* 현재 SQL문이 위치한 행번호 */
  long   orahoc;                 /* 요청한 MAXOPENCURSORS 값 */
  long   oramoc;                 /* 최대로 open할 수 있는 커서 개수 */
  long   oracoc;                 /* 현재 사용되고 있는 커서 개수 */
  long   oranor;                 /* 재할당된 커서 캐쉬의 개수 */
  long   oranpr;                 /* 파싱된 SQL문의 개수 */
  long   oranex;                 /* 실행된 SQL문의 개수 */
 };
```

ORACA를 사용한 예는 아래와 같다.

```
#include <sqlca.h>
#include <oraca.h>
…
main()
{
        …
        char SQLSTATE[6];
        EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error");
        EXEC SQL CONNECT :userid;
        EXEC ORACLE OPTION (ORACA=YES);
        oraca.oradbgf = 1; /* debug 연산 활성화 */
        oraca.oracchf = 1; /* 커서 캐시 통계 정보 수집 */
        oraca.orastxtf = 3; /* 항상 SQL문 저장 */
        printf("Enter department number: ");
        …
);

void sql_error(char *errmsg)
{
        char buf[6];
```

```
        strcpy(buf, SQLSTATE);
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL COMMIT WORK RELEASE;
        if (strncmp(errmsg, "Oracle error", 12) == 0)
                printf("₩n%s, sqlstate is %s₩n₩n", errmsg, buf);
        else
                printf("₩n%s₩n₩n", errmsg);
        printf("Last SQL statement: %.*s₩n",
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
        printf("₩nAt or near line number %d₩n", oraca.oraslnr);
        printf("₩nCursor Cache Statistics₩n----------------------₩n");
        printf("Maximum value of MAXOPENCURSORS: %d₩n", oraca.orahoc);
        printf("Maximum open cursors required: %d₩n", oraca.oramoc);
        printf("Current number of open cursors: %d₩n", oraca.oracoc);
        printf("Number of cache reassignments: %d₩n", oraca.oranor);
        printf("Number of SQL statement parses: %d₩n", oraca.oranpr);
        printf("Number of SQL statement executions: %d₩n", oraca.oranex);
        exit(1);
}
...
```

3.7 오라클 접속 및 해제

3.7.1 접속

오라클에 질의를 하거나 데이타를 조작하기 전에 데이타베이스에 접속을 해야한다. 접속을 하기 위해 CONNECT 문을 사용한다.
아래는 간단하게 접속을 하는 방법을 보여 준다. 처음 예에서 username이나 password는 char 또는 varchar 호스트 변수이다. 밑의 예에서 usr_pwd는 사용자 명과 암호를 “scott/tiger”처럼 슬래쉬(/)로 구분해서 포함해야 한다.

```
EXEC SQL CONNECT :username IDENTIFIED BY :password ;
또는
EXEC SQL CONNECT :usr_pwd ;
```

아래의 예는 오라클의 사용자 scott(암호는 tiger)로 접속하는 코드이다.

```
char *username = "SCOTT";
char *password = "TIGER";
...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

여기서 CONNECT 문에 문자열을 직접 주면 에러가 발생한다.

```
EXEC SQL CONNECT 'scott' IDENTIFIED BY 'tiger';    /* 에러 발생!!! */
```

3.7.2 원격에서 접속하는 경우

원격에서 오라클 서버를 접속하려면(예를 들어 Pro*C/C++ 프로그램은 db.snu.ac.kr에 있고, 오라클 서버는 dbpub.snu.ac.kr에 있는 경우), 먼저 tnsname.ora 파일을 편집한다. tnsname.ora 는 $ORACLE_HOME/network/admin (리눅스에서 설치한 경우)밑에 위치한다.

```
   dbpub =
     ( DESCRIPTION =
        (ADDRESS = (PROTOCOL = TCP)(HOST = dbpub.snu.ac.kr)(PORT = 1521))
        (CONNECT_DATA = (SERVICE_NAME = ora9i))
     )
```

%주의 : 이 설정은 서버마다 다르기 때문에, 관리자가 알려준 사항을 따라야 한다.

원격에서 접속할 때 CONNECT 문에 옵션을 하나 추가한다.

```
/* 호스트 변수 설정 */
char username[10] = "scott";
char password[10] = "tiger";
char db_string[20] = "dbpub";
…
/* 원격 DB에 접속*/
EXEC SQL CONNECT :username IDENTIFIED BY :password USING :db_string;
```

3.7.3 접속 종료

오라클과의 접속을 해제할 경우에는 다음과 같이 두 가지 문장을 사용할 수 있다.

① EXEC SQL COMMIT[1] WORK RELEASE;
② EXEC SQL ROLLBACK WORK RELEASE;

첫 번째 문장은 프로그램 내의 SQL문을 commit하고 접속을 해제하는 경우이고 두 번째 문장은 rollback한 뒤 접속을 해제하는 경우이다. RELEASE는 자원을 해제하고 log off하라는 의미이다. 일반적으로 프로그램이 정상적으로 수행된 경우에는 첫 번째 문장을 이용하여 접속을 끊고, 에러 발생시 프로그램이 종료되는 경우에는 두 번째 문장으로 접속을 끊는다.

3.8 컴파일 및 실행

sample1.pc 파일을 proc를 이용하여 예비 컴파일 한다.

```
$proc sample1.pc
```

예비 컴파일 하면 sample1.c 파일이 생성된다. sample1.c 파일을 gcc를 이용하여 컴파일 한다. 이 때 오라클 관련 라이브러리들을 포함한다.

```
$gcc -o sample1 -I$ORACLE_HOME/precomp/public -L$ORACLE_HOME/lib -lclntsh sample1.c
```

-I(대문자 i)는 gcc의 옵션으로 헤더 파일의 디렉토리 경로를 지정한다. 여기서는 sqlca.h와 oraca.h 등의 오라클 관련 헤더 파일들의 위치한 $ORACLE_HOME/precomp/public을 포함하도록 하고 있다.
-L은 gcc의 옵션으로 라이브러리가 위치한 디렉토리 경로를 지정한다. 여기서는 $ORACLE_HOME/lib를 포함하고 있다.
-l(소문자 L)은 gcc의 옵션으로 라이브러리를 지정한다. 여기서는 clntsh 라이브러리를 가리킨다. $ORACLE_HOME/lib 밑에 있는 libclntsh.so 라는 공유 라이브러리(shared library)를 포함하라는 의미이다.

컴파일이 끝나면, sample1이라는 실행 파일이 생성된다.

[1] COMMIT과 ROLLBACK은 트랜잭션에서 나오는 용어이다. 트랜잭션의 의미는 DB 책들을 참고한다. COMMIT과 ROLLBACK에 대한 자세한 사용법은 ProC.pdf의 3장을 참고한다.

IV. 예제 설명

4.1 Pro*C 예제

오라클에서 제공하는 sample3.pc 파일을 설명한다. 이 예제는 커서의 사용법을 보여 주고 있다.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>

#define NAME_LENGTH     8
#define ARRAY_LENGTH    5

/* 사용자 명 */
char *username = "SCOTT";
char *password = "TIGER";

/*  호스트 구조체의 선언 */
struct
{
    int     emp_number[ARRAY_LENGTH];
    char    emp_name[ARRAY_LENGTH][NAME_LENGTH];
    float   salary[ARRAY_LENGTH];
} emp_rec;

void print_rows(int n)
{
    int i;

    printf("₩nNumber   Employee   Salary");
    printf("₩n------   --------   -------₩n");
```

```
    for (i = 0; i < n; i++)
        printf("%d      %s   %8.2f₩n", emp_rec.emp_number[i],
                emp_rec.emp_name[i], emp_rec.salary[i]);

}

void sql_error(char* msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("₩n%s", msg);
    printf("₩n% .70s ₩n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT_FAILURE);
}

void main()
{
    int   num_ret;                  /* 반환된 행의 수 */

    /* 오라클에 접속 */
    EXEC SQL WHENEVER SQLERROR DO sql_error("Connect error:");

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("₩nConnected to ORACLE as user: %s₩n", username);

    EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error:");
    /* 커서 선언 */
    EXEC SQL DECLARE c1 CURSOR FOR
        SELECT empno, ename, sal FROM emp;

    EXEC SQL OPEN c1;

    num_ret = 0;    /* 초기하 */
```

```
    /* fetch 루프 */
    EXEC SQL WHENEVER NOT FOUND DO break;
    for (;;) {
        EXEC SQL FETCH c1 INTO :emp_rec;

        print_rows(sqlca.sqlerrd[2] - num_ret);
        num_ret = sqlca.sqlerrd[2];          /* num_ret 재 설정. */
    }

    /*  마지막 ftech후 남아있는 행이 있다면, 남아있는 행들을 출력 */
    if ((sqlca.sqlerrd[2] - num_ret) > 0)
        print_rows(sqlca.sqlerrd[2] - num_ret);

    EXEC SQL CLOSE c1;
    printf("₩nAu revoir.₩n₩n₩n");

    /*  접속 해제 */
    EXEC SQL COMMIT WORK RELEASE;
    exit(EXIT_SUCCESS);
}
```