SQL Embebido

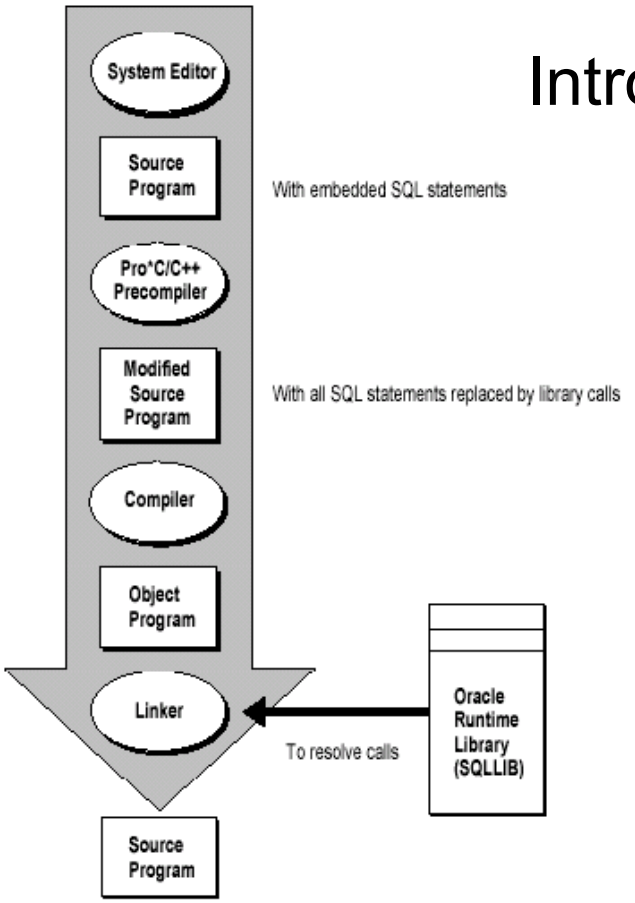
Introducción (I)

- **SQL directo** (o interactivo): las instrucciones se ejecutan directamente desde el terminal y el resultado de las consultas se visualiza en el monitor de la estación.
- **SQL embebido**: utilizado intercalando sus instrucciones en el código de un programa escrito en un lenguaje de programación al que se denomina *lenguaje anfitrión* (FORTRAN, COBOL, C, etc.).

Introducción (II)

- La forma de construir un programa con SQL embebido varía dependiendo del lenguaje y el gestor de BD utilizados
- Nosotros estudiaremos el caso de SQL embebido en ***C*** contra ***Oracle***.
- Oracle ha definido un lenguaje especial para soportar este tipo de programación: lenguaje **Pro*C**

Introducción (III)



- SQL embebido en Oracle: Proceso de compilación y enlazado (link) de un programa Pro*C

Introducción (IV)

- Estructura de un programa Pro*C:

```
Inclusión de cabeceras

Declaración de variables
    Declaración de VARIABLES huésped

Comienzo del código del programa
        ...
        Instrucciones propias del lenguaje
            ...
        CONEXIÓN con la base de datos
            ...

             Instrucciones de SQL: DML, DDL
                                 +
             Instrucciones propias del lenguaje
            ...

        DESCONEXIÓN de la base de datos
            ...
        Instrucciones propias del lenguaje
            ...

Fin del código de programa
```


Índice:

- Sintaxis general de las instrucciones básicas de SQL embebido
- Sentencias básicas de SQL embebido:
  - Declaración de variables
  - Control de errores
  - Conexión /desconexión de la base de datos
  - Consultas (sentencias *select*)
  - Sentencias de modificación de datos: (*insert, update,delete*)
  - Sentencias de modificación de la BD (create, …)
  - Ejecución de sentencias SQL a partir de *strings*: SQL *dinámico*.

Sintaxis general de las sentencias de SQL embebido (I)

- En general, cualquier instrucción del SQL interactivo se puede utilizar en el SQL embebido con pequeñas variaciones sintácticas.
- Existen, además, instrucciones específicas del SQL para la gestión de los conjuntos de tuplas obtenidos como resultado de alguna consulta.
- También: directivas del precompilador: gestión de errores, delimitación de segmentos de programa, etc.

Sintaxis general de las sentencias de SQL embebido (II)

- Consideraciones generales de sintaxis:
  - Todas las instrucciones del SQL embebido van precedidas de las palabras reservadas EXEC SQL y finalizan con un símbolo especial.
  - En el lenguaje Pro*C este símbolo es el punto y coma (;).

```
EXEC SQL delete from emp;
```

  - Esta notación permite al precompilador identificar y procesar estas instrucciones para traducirlas a código C estándar.

Declaración de variables (I)

- Las variables del lenguaje que son utilizadas en instrucciones SQL (variables *huéspedes*) deben ser declaradas en una sección especial encabezada y terminada de la siguiente forma:

```
EXEC SQL BEGIN DECLARE SECTION;
   ...
EXEC SQL END DECLARE SECTION;
```

- Las variables huésped constituyen la única vía de intercambio de datos entre el programa huésped y la base de datos.

Declaración de variables (II)

- La declaración y el uso de una variable huésped se realizan en la forma convencional del lenguaje C:

```
long empno;
…
empno = 123;
```

- Pueden aparecer en una instrucción SQL en cualquier lugar en el que puede aparecer una expresión o valor del mismo tipo, y deben ir precedidas del símbolo *dos puntos* (:).

```
EXEC SQL SELECT * … FROM EMP WHERE EMPNO = :empno;
```

- No se pueden utilizar para representar a objetos de la base de datos (nombres de tabla, columna, etc.)

```
char tabla[] = "emp";
…
EXEC SQL SELECT * FROM :tabla
```

(INCORRECTO!!!)

Declaración de variables (III)

- Los siguientes tipos utilizados en C pueden ser utilizados para definir variables huésped:

char	*Carácter Simple*
char[n]	*Cadena de n caracteres (string)*
int	*Valor entero*
short	*Entero corto*
long	*Entero Largo*
float	*Valor de punto flotante (precisión sencilla)*
double	*Valor de punto flotante (precisión doble)*
VARCHAR[n]	*Cadena de caracteres de longitud variable*

- Las variables deben tener un tipo apropiado al uso que se va a hacer de ellas.

Declaración de variables (IV)

• Caso especial: Pseudotipo VARCHAR. Permite almacenar valores del tipo VARCHAR[n] de SQL.

• Declaraciones VARCHAR traducidas por el precompilador de *Oracle* a un tipo estándar de *C*: una estructura

```
VARCHAR nombre[10]; ⇒  Struct{
                       int len;        /* longitud válida del string */
                       char arr[10];   /* texto del string */
                           } nombre;
```

• Para imprimir un tipo VARCHAR:

```
printf("%.*s", nombre.len, nombre.arr);
```

Ejemplo 1 - Enunciado

- Programa que imprime el nombre y comisión de un empleado a partir de su identificador (numero de empleado).

```
Name          Null?          Type
---------     -----------    ---------------
 EMPNO        NOT NULL       NUMBER(4)
 ENAME        NOT NULL       VARCHAR2(10)
 JOB          NOT NULL       VARCHAR2(9)
 MGR          NOT NULL       NUMBER(4)
 HIREDATE     NOT NULL       DATE
 SAL          NOT NULL       NUMBER(7,2)
 COMM                        NUMBER(7,2)
 DEPTNO       NOT NULL       NUMBER(2)
```


Ejemplo 1 - Declaración de variables

```
/**** Ejemplo1 ****/

/* Inclusión de cabeceras usuales en C */
#include <stdio.h>
#include <string.h>


/* Declaración de variables "huésped", que serán utilizadas en
sentencias SQL */

EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR nomemp[10];
    float comision;
    long num;
EXEC SQL END DECLARE SECTION;
```

Control de errores (I)

- Mediante la sentencia WHENEVER:

```
EXEC SQL WHENEVER SQLERROR DO GestionError();
```

- Formato general:

	(Condición)	(Acción)
WHENEVER	SQLERROR	DO BREAK
	SQLWARNING	DO Funcion()
	NOT FOUND	CONTINUE
		~~GOTO Etiqueta~~

Control de errores (II)

- Condiciones:
  - **SQLERROR** : Se ha producido un error de SQL.
    Ejemplos: error de conexión, violación de una restricción, etc.
  - **SQLWARNING** : Hay un *warning* o aviso.
    Ejemplos: truncado de datos
  - **NOT FOUND** : Ya hemos recuperado todas las tuplas del resultado de la consulta (usando un cursor).
- Acciones:
  - **DO BREAK** : Si está en un bucle sale de él
  - **DO Funcion()** : Ejecuta la función que se le indica
  - **CONTINUE** : Continúa, ignorando el error
  - **GOTO Etiqueta** : Va la etiqueta indicada

Control de errores (III)

- Cuando se ejecuta una sentencia WHENEVER, tiene efecto hasta el final del programa o hasta que la siguiente sentencia WHENEVER.
- Ejemplo:

```
EXEC SQL WHENEVER SQLERROR DO GestionError();
  . . . Sentencias
  . . . En esta parte, si hay un error de SQL se
  llama a la función GestionError
EXEC SQL WHENEVER SQLERROR CONTINUE;
  . . . A partir de aquí, los errores SQL se ignoran
```


Control de errores (IV)

- La sentencia WHENEVER se usa a veces en conjunción con la estructura SQLCA (SQL Communications Area): guarda el estado de la última ejecución de una sentencia SQL:

```
struct sqlca{
         char      sqlcaid[8];
         long      sqlabc;
         long      sqlcode;
         struct{
                   unsigned short  sqlerrml;
                   char            sqlerrmc[70];
                   } sqlerrm;
         char      sqlerrp[8];
         long      sqlerrd[8];
         char      sqlwarn[8];
char     sqlext[8];
};

struct sqlca sqlca;
```

  - Sqlcode: Código de resultado. Es 0 si fue correcta
  - Sqlerrm.sqlerrmc: Mensaje de error
  - sqlerrd[2]: Número de filas (tuplas) afectadas por la última sentencia
- Necesario incluir el fichero de cabecera <sqlca.h> para poder utilizar la variable *sqlca*.

Ejemplo 1 – Con control de errores

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

/* Cabeceras de SQL Communication Area, para control de errores, etc. */
#include <sqlca.h>

/* Prototipos de funciones */
void GestionErrores();

/* Declaración de variables "huésped", que serán utilizadas en sentencias SQL */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR nomemp[10];    float comision;    long num;
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR DO GestionError();

main(){
 ...
}

/* En caso de error: mostrar el código y mensaje de error,
   se aborta la transacción y se sale del programa          */
void GestionError(){
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("Error %d\n",sqlca.sqlcode);
  printf("Error %s\n",sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```


Conexión/Desconexión

- Antes de efectuar cualquier operación sobre la BDD es necesario conectarse a la misma:

  EXEC SQL CONNECT *usuario/pasword;*

  EXEC SQL CONNECT *usuario* IDENTIFIED BY *password*

- Asimismo, para liberar los recursos asociados a la sesión, es obligatorio desconectarse de la base de datos antes de que el programa finalice:

  EXEC SQL COMMIT [WORK] RELEASE;

- Si se desea terminar deshaciendo la última transacción, la desconexión se hace de este modo:

  EXEC SQL ROLLBACK [WORK] RELEASE;

Ejemplo 1 - Conexión

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

/* Prototipos de funciones */
void GestionError();


EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR nomemp[10];
    float comision;
    long num;
    char usuario [20] = "scott/tiger";
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR DO
GestionError();
```

```
main(){

  EXEC SQL CONNECT :usuario;
  ...
  EXEC SQL COMMIT RELEASE;
}
/* Errores */

void GestionError(){
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("Error %d\n",sqlca.sqlcode);
  printf("Error %s\n",sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK RELEASE;
  exit(1);
}
```


Consultas (Selección de datos)

- Las consultas en SQL embebido, aunque similares a sus equivalentes en SQL interactivo, tienen algunas particularidades.

  "Obtener el nombre del empleado cuyo número es 7900"

- En SQL interactivo:

```
SELECT ENAME
  FROM EMP
  WHERE EMPNO=7900
```

  Resultado:

```
ENAME
---------
JAMES
```

- Con SQL embebido, debemos indicar dónde guardar el valor obtenido por la consulta, en vez de ser mostrado por pantalla.
- Se emplearán las variables huésped definidas en la `DECLARE SECTION`.

Consultas: Una fila (I)

- La operación de consulta se realiza con la siguiente variante de la sentencia SELECT del lenguaje SQL interactivo:

```
SELECT [ALL | DISTINCT] lista_selección
 INTO lista_variables
 FROM lista_relación
 [WHERE condición_búsqueda]
 [GROUP BY lista_atributo]
 [HAVING condición_búsqueda]
```

- La sentencia SELECT debe devolver **exactamente una fila**, en caso contrario se producirá un error.
- La cláusula **INTO** especifica la lista de variables que almacenarán los datos seleccionados.
- Los elementos de la *lista_selección* y de la *lista_variables* se corresponden uno a uno en el orden en que aparecen
  - Deben tener el mismo número de elementos y ser de tipos de datos compatibles.

Ejemplo 1

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

void GestionError();

EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR nomemp[10];
    float comision;
    long num;
    char usuario [20]
       = "scott/tiger";
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR DO
GestionError();
```

```
main()
{

EXEC SQL CONNECT :usuario;

printf (Numero?"); scanf ("%i", &num);
EXEC SQL SELECT ENAME, COMM
         INTO :nomemp, :comision
         FROM EMP
         WHERE EMPNO=:num;
printf("%.*s, con comision %f", nomemp.len,
nomemp.arr, comision);

EXEC SQL COMMIT RELEASE;
}

void GestionError()
{
 printf("Error %d\n",sqlca.sqlcode);
 printf("Error %s\n",sqlca.sqlerrm.sqlerrmc);
 EXEC SQL ROLLBACK RELEASE;
 exit(1);
}
```

Consultas: Una fila (II)

- Posibles errores

```
EXEC SQL SELECT ENAME, COMM
INTO :nomemp, :comision
FROM EMP WHERE EMPNO>:num;
```

¿empleados cuyo número es mayor quee 1000?:

```
ORA-01422: la recuperación exacta devuelve un
  número mayor de filas que el solicitado
```

¿empleados cuyo número es mayor que 9999?:

```
ORA-01403: no se han encontrado datos
```


Consultas: Cursores

- El concepto de *cursor* nos permite simular que la consulta que se ejecuta es un fichero secuencial, que abrimos, vamos leyendo hasta el final, y cerramos. Esto nos permite recuperar los datos de una consulta que devuelva (cero, una o) más de una tupla.

- Para usar un cursor, se seguirán los siguientes pasos:
  - Declaración del cursor, asociándolo a una consulta

```
EXEC SQL DECLARE nombre_cursor CURSOR FOR
   Sentencia_select;
```

  - Abrir el cursor

```
EXEC SQL OPEN nombre_cursor;
```

  - Dentro de un bucle, obtener secuencialmente todas las tuplas que devuelve la consulta

```
EXEC SQL FETCH nombre_cursor INTO
   variables_anfitrionas;
```

  - Cerrar el cursor

```
EXEC SQL CLOSE nombre_cursor;
```

Ejemplo

```
…
main()
{
EXEC SQL CONNECT :usuario;

printf (“Numero?”); scanf (“%i”, &num);

EXEC SQL DECLARE empleados CURSOR FOR
            SELECT ENAME, COMM
                  FROM EMP
                  WHERE EMPNO>:num;
EXEC SQL OPEN empleados;
EXEC SQL WHENEVER NOT FOUND DO break;
while(1){
         EXEC SQL FETCH empleados  INTO :nomemp, :comision;
         printf("%.*s - %f\n", nomemp.len, nomemp.arr, comision);
}

printf("\nSe han obtenido %d empleados cuyo numero es >1000\n",
sqlca.sqlerrd[2]);

EXEC SQL CLOSE empleados;
EXEC SQL COMMIT RELEASE;
}
```


Control de valores nulos (I)

- Otro problema a resolver: ¿qué sucede cuando alguno de los valores que recuperamos en una variable huésped es nulo?
- En nuestro “Ejemplo 1”, un empleado puede no tener asignada ninguna comisión.

```
$ ./ejemplo
  Numero de empleado?7369
```

- La ejecución del programa dará lugar a un error:

```
Error -1405
Error ORA-01405: el valor de la columna recuperada es NULL
```

Control de valores nulos (II)

- Detección de valores nulos: mediante *variables indicador*
- Cada variable huésped puede tener asociada una variable indicador, que indica si el valor almacenado está **truncado** o es **nulo**
- Las variables indicador deben definirse en la DECLARE SECTION, y son de tipo *short*
- La asociación con la variable huésped correspondiente se realiza en cada sentencia sql embebida:
- Notación (2 alternativas):
  - :variable_huésped INDICATOR :variable_indicador
  - :variable_huésped:variable_indicador

Control de valores nulos (III)

- Valores posibles de la variable indicador:
  - -1: la variable huésped asociada contiene un valor nulo
  - 0: la variable huésped contiene un valor correcto e intacto
  - >0: la variable huésped contiene un valor que ha sido truncado al ser asignado a la variable
  - -2: la variable huésped contiene un valor truncado que no pudo ser recuperado correctamente de la base de datos.

Ejemplo

```
EXEC SQL BEGIN DECLARE SECTION;
    …
    float comision;
    short comision_ind;
    …
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empleados CURSOR FOR
       SELECT ENAME, COMM
           FROM EMP
           WHERE EMPNO>:num;

EXEC SQL OPEN empleados;

EXEC SQL WHENEVER NOT FOUND DO break;
while(1){
    EXEC SQL FETCH empleados INTO :nomemp, :comision:comision_ind;
    if (comision_ind==0)
       printf("%.*s \t- %i\n", nomemp.len, nomemp.arr,comision);
    else
       printf("%.*s \t- No disponible\n", nomemp.len, nomemp.arr);
}
EXEC SQL CLOSE empleados;
```

```
Numero de empleado?1000

SMITH    - No disponible
ALLEN    - 0
WARD     - 0
JONES    - No disponible
MARTIN   - 0
BLAKE    - No disponible
CLARK    - No disponible
SCOTT    - No disponible
KING     - No disponible
TURNER   - 0
ADAMS    - No disponible
JAMES    - No disponible
FORD     - No disponible
MILLER   - No disponible
```


Modificación de la BD

- El programa puede ejecutar sentencias DDL y DML en SQL:
  - EXEC SQL INSERT ... INTO
  - EXEC SQL DELETE FROM ... WHERE
  - EXEC SQL UPDATE ... SET ... WHERE

  } DML

  - EXEC SQL CREATE TABLE …
  - EXEC SQL ALTER TABLE …
  - …

  } DDL

Ejemplo Borrado

```
EXEC SQL BEGIN DECLARE SECTION;
    long codigo;
    char usuario[20];
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR DO GestionError();

main()
{

strcpy(usuario,"scott/tiger");
EXEC SQL CONNECT :usuario;
codigo = 7900;
EXEC SQL DELETE FROM EMP WHERE EMPNO = :codigo;
EXEC SQL COMMIT RELEASE;
}
```


Ejemplo inserción (con nulos)

```
EXEC SQL BEGIN DECLARE SECTION;
    long EMPNO;  VARCHAR ENAME [10] ;       VARCHAR JOB [9];
    long MGR;    VARCHAR HIREDATE[11];      double SAL;
    double COMM; short COMM_ind;
    int DEPTNO;
EXEC SQL END DECLARE SECTION;
…
EMPNO=7000;
strcpy (ENAME.arr, "Pedro"); ENAME.len=5;
strcpy (JOB.arr, "MANAGER"); JOB.len=7;
MGR=7527;
Strcpy (HIREDATE.arr,"03-MAY-03"); HIREDATE.len=11);
SAL = 50000.0;
COMM=0;
COMM_IND=-1;
DEPNO=10;

EXEC SQL INSERT INTO EMP
 (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPNO)
VALUES
 (:EMPNO,:ENAME,:JOB,:MGR,:HIREDATE,:SAL,:COMM:COMM_ind,:DEPTNO);
…
```

Ejemplo Creación Tabla

```
#include <stdio.h>
#include <string.h>
#include <sqlca.h>
void GestionError();
EXEC SQL WHENEVER SQLERROR DO GestionError();
main()
{
char nom[20];
strcpy(nom,"scott/tiger");
EXEC SQL CONNECT :nom;
EXEC SQL create table prueba (
         nombre char(10),
         num number(4)
        );
EXEC SQL COMMIT RELEASE;
}
void GestionError()
{
 printf("error %d\n",sqlca.sqlcode);
 printf("error %s\n",sqlca.sqlerrm.sqlerrmc);
 EXEC SQL ROLLBACK RELEASE;
 exit(1);
}
```


Ej. 2: estructuras como variables huésped

```
# include <stdio.h>
# include <string.h>
exec sql include sqlca;

exec sql begin declare section;

  struct pro_ {
      char cod_pro[4];
      char nombre[41];
      int telefono;
      char cod_dep[6];
  } pro;

  char cd[6];

exec sql end declare section;
```

```
main(){
  char usuario[20];
  exec sql declare c_pro cursor for
  select cod_pro, nombre, telefono, cod_dep
  from Profesor
  where cod_dep = :cd;
  printf("Dame el código del departamento ");
  scanf("%5s", cd);
  printf("%-4s %-40s %-10s", "Código", "Nombre",
         "Teléfono");
  printf("\n---------------------------------------\n");
  strcopy(usuario, "scott/tiger");
  exec sql connect :usuario;
  exec sql open c_pro;
  exec sql fetch c_pro into :pro;

  while (sqlca.sqlcode == 0)
  {
   printf("%-4s %-40.40s %-5d", pro.cod_pro, pro.nombre,
          pro.telefono);
   printf("\n");
   exec sql fetch c_pro into :pro;
  }
  exec sql close c_pro;
  exec sql disconnect;
}
```

SQL Dinámico (I)

- **SQL embebido estático:** sentencias codificadas directamente en el programa
- **SQL embebido dinámico:** se construye una cadena de caracteres (string) y se ejecuta como consulta.
- En selección de datos: necesitamos usar cursores
  1. Construir el string, utilizando funciones C (strcpy, strcat, etc.).
  2. Preparar la sentencia SQL a partir del string (`PREPARE consulta FROM string`).
  3. Utilizar el cursor de la forma indicada anteriormente, para esa consulta (`DECLARE nombre_cursor CURSOR FOR consulta`).

Ejemplo

```
main()
{
 EXEC SQL BEGIN DECLARE SECTION;
   char usuario[20], st[100];
 EXEC SQL END DECLARE SECTION;

 strcpy(usuario,"scott/tiger");
 EXEC SQL CONNECT :usuario;
 strcpy(st, "SELECT ename from emp where empno=7900");

 EXEC SQL PREPARE A FROM :st;
 EXEC SQL DECLARE C CURSOR FOR A;
 EXEC SQL OPEN C;
 EXEC SQL FETCH C INTO :nombre;
 EXEC SQL CLOSE C;
 printf("%s",nombre.arr);
 EXEC SQL COMMIT RELEASE;
}
```

SQL Dinámico (II)

- En el caso de que sea una sentencia de modificación se realiza utilizando `EXECUTE IMMEDIATE`.
- Creación de la tabla “tbl”

```
EXEC SQL EXECUTE IMMEDIATE
   "CREATE TABLE tbl (col1 VARCHAR(4))";
```

- Para insertar un valor en esa tabla:

```
strcpy(st, "INSERT INTO tbl values ('aaaa')");
EXEC SQL EXECUTE IMMEDIATE :st;
```


SQL Dinámico (III)

- Si hay variables anónimas en la cadena que construye el cursor se utiliza:

```
EXEC SQL OPEN nombre_cursor USING :variables_host
```

Ejemplo

```
strcpy(st, "SELECT ename from emp where empno=:anonima");
EXEC SQL PREPARE consulta FROM :st;
EXEC SQL DECLARE Curs CURSOR FOR consulta;
scanf(“%i”,&num);
EXEC SQL OPEN Curs USING :num;
EXEC SQL FETCH Curs INTO :nombre;
EXEC SQL CLOSE Curs;
printf("%s",nombre.arr);
```