CHAPTER

3

Embedded SQL

Oracle SQL, introduced in the previous chapter, is not a language that can be used to build sophisticated database applications, but it is a very good language for defining the structure of the database and generating ad hoc queries. However, to build applications, the power of a full-fledged high-level programming language is needed. Embedded SQL provides such an environment to develop application programs. The basic idea behind embedded SQL is to allow SQL statements in a program written in a high-level programming language such as C or C++. By embedding SQL statements in a C/C++ program, one can now write application programs in C/C++ that interact (read and write) with the database. Oracle provides a tool, called Pro*C/C++, which allows for applications to be developed in the C or C++ language with embedded SQL statements. The Pro*C/C++ preprocessor parses the embedded program and converts all SQL statements to system calls in C/C++ and produces a C/C++ program as its output. This C/C++ program can be compiled in the usual manner to produce the executable version of the application program. This chapter introduces concepts and techniques needed to write successful embedded SQL programs in Oracle using the C or C++ language. Most of the concepts are introduced using the C language. A separate section is devoted to C++ programs.

3.1 Host Variables

Since SQL statements are to be embedded within the C program, there is a need for a mechanism to pass values between the C program environment and the SQL statements that communicate with the Oracle database server. Special variables, called *host variables*, are defined in the embedded program for this purpose. These host variables are defined between the `begin declare section` and `end declare section` directives of the preprocessor as follows:

```
EXEC SQL begin declare section;
  int     cno;
  varchar cname[31];
  varchar street[31];
  int     zip;
  char    phone[13];
EXEC SQL end declare section;
```

The data types of the host variables must be compatible with the data types of the columns of the tables in the database. Figure 3.1 shows data types in C that are compatible with commonly used data types in Oracle. The `char` data type in Oracle is mapped to the `char` data type in C. The `char(N)` data type in Oracle is mapped to an array of characters in C. Notice that the size of the C array is one more than the size of the character string in Oracle. This is due to the fact that C character strings require an additional character to store the end-of-string character (`\0`). Oracle's Pro*C preprocessor provides a `varchar` array data type in C that corresponds to the `varchar(N)` Oracle data type. Again, the size of the C `varchar` array is one more than the maximum size of the `varchar` string of Oracle. The `varchar` array in C is declared as

```
varchar cname[31];
```

and the Pro*C preprocessor produces the following C code corresponding to the above declaration:

```
/* varchar cname[31]; */
struct {
  unsigned short len;
  unsigned char arr[31];
} cname;
```

Note that the `varchar` array variable `cname` has been transformed into a structure (with the same name) containing two fields: `arr` and `len`. The `arr` field will store the actual string and the `len` field will store the length of the character string. When sending a `varchar` value to the database, it is the responsibility of the programmer to make sure that both fields, `arr` and `len`, are assigned proper values. When receiving such a value from the database, both fields are assigned appropriate values by the system. The `date` data type is mapped to a fixed-length (10 characters, corresponding to the default `date` format in Oracle) character string in C. The

Figure 3.1 Compatible Oracle and C data types.

Oracle Data Type	C Data Type
`char`	`char`
`char(N)`	`char array[N+1]`
`varchar(N)`	`varchar array[N+1]`
`date`	`char array[10]`
`number(6)`	`int`
`number(10)`	`long int`
`number(6,2)`	`float`

numeric data types are appropriately mapped to `small int`, `int`, `long int`, `float`, or `double`, depending on their precisions in Oracle.

The host variables are used in the usual manner within C language constructs; however, when they are used in the embedded SQL statements, they must be preceded by a colon (`:`). Some examples of their usage are shown in the following code fragment.

```
scanf("%d",&cno);
EXEC SQL select cname
         into   :cname
         from   customers
         where  cno = :cno;

scanf("%d%s%s%d%s",&cno,cname.arr,street.arr,&zip,phone);
cname.len  = strlen(cname.arr);
street.len = strlen(street.arr);
EXEC SQL insert into customers
         values (:cno,:cname,:street,:zip,:phone);
```

The select statement in the above example has an additional clause, the `into` clause, which is required in embedded SQL since the results of the SQL statements must be stored someplace. The `select into` statement can be used only if it is guaranteed that the query returns exactly one or zero rows. A different technique is used to process queries that return more than one row. This technique, which uses the concept of a *cursor*, is discussed in Section 3.5. Note that all occurrences of the host variables within the embedded SQL statements are preceded by a colon. Also, the `len` fields of all `varchar` arrays in C are set to the correct lengths before sending the host variables to the database.

3.2 Indicator Variables

A `null` value in the database does not have a counterpart in the C language environment. To solve the problem of communicating `null` values between the C program and Oracle, embedded SQL provides *indicator variables*, which are special integer variables used to indicate if a `null` value is retrieved from the database or stored in the database. Consider the `orders` table of the mail-order database. The following is the declaration of the relevant host and indicator variables to access the `orders` table.

```
EXEC SQL begin declare section;
struct {
    int     ono;
    int     cno;
    int     eno;
    char    received[12];
    char    shipped[12];
  } order_rec;
  struct {
    short     ono_ind;
    short     cno_ind;
    short     eno_ind;
    short     received_ind;
    short     shipped_ind;
  } order_rec_ind;
  int  onum;
EXEC SQL end declare section;
```

The code below reads the details of a particular row from the `orders` table into the host variables declared above and checks to see whether the `shipped` column value is `null`. A `null` value is indicated by a value of −1 for the indicator variable. The database server returns a value of 0 for the indicator variable if the column value retrieved is not `null`.[1]

```
scanf("%d",&onum);
EXEC SQL select *
         into   :order_rec indicator :order_rec_ind
```

1. The indicator variable is also used for other purposes—for example, it is used to indicate the length of a string value that was retrieved from the database and that was truncated to fit the host variable into which it was retrieved.

```
         from   orders
         where  ono = :onum;
if (order_rec_ind.shipped_ind == -1)
  printf("SHIPPED is Null\n");
else
  printf("SHIPPED is not Null\n");
```

To store a `null` value into the database, a value of −1 should be assigned to the indicator variable and the indicator variable should be used in an update or insert statement. For example, the following code sets the `shipped` value for the order with order number `1021` to `null`.

```
onum = 1021;
order_rec_ind.shipped_ind  = -1;
EXEC SQL update orders
         set shipped = :order_rec.shipped indicator
                       :order_rec_ind.shipped_ind
         where ono = :onum;
```

Notice that the `order_rec.shipped` value is undefined, because it will be ignored by the database server.

3.3 SQL Communications Area (sqlca)

Immediately after the Oracle database server executes an embedded SQL statement, it reports the status of the execution in a variable called `sqlca`, the SQL communications area. This variable is a structure with several fields, the most commonly used one being `sqlcode`. Typical values returned in this field are shown below.

`sqlca.sqlcode`	Interpretation
0	SQL statement executed successfully
> 0	No more data present or values not found
< 0	Error occurred while executing SQL statement

To include the `sqlca` definition, the following statement must appear early in the program:

```
EXEC SQL include sqlca;
```

Here are two code fragments that illustrate the use of `sqlca.sqlcode`.

Error check: Consider the following code fragment, which attempts to add a new row into the `customers` table.

```
EXEC SQL set transaction read write;
EXEC SQL insert into customers values
         (custseq.nextval,:customer_rec.cname,
          :customer_rec.street,:customer_rec.zip,
          :customer_rec.phone);
if (sqlca.sqlcode < 0) {
  printf("\n\nCUSTOMER (%s) DID NOT GET ADDED\n",
         customer_rec.cname.arr);
  EXEC SQL rollback work;
  return;
}
EXEC SQL commit;
```

After starting a transaction to read and write to the database,[2] this program fragment attempts to insert a new row into the `customers` table using the `EXEC SQL insert into` statement. There could be several reasons why this statement may not execute successfully, among them primary key constraint violation, data type mismatches, and wrong number of columns in the insert statement. The value of `sqlca.sqlcode` is checked to see if it is less than 0. If an error is indicated, a message is sent to the user and the transaction is rolled back. Otherwise, the transaction is committed and the row is successfully inserted.

`Not found` check: Consider the following code fragment:

```
EXEC SQL select zip, city
         into   :zipcode_rec
         from   zipcodes
         where  zip = :customer_rec.zip;
if (sqlca.sqlcode > 0) {
  zipcode_rec.zip = customer_rec.zip;
  printf("Zip Code does not exist; Enter City: ");
  scanf("%s",zipcode_rec.city.arr);
  zipcode_rec.city.len = strlen(zipcode_rec.city.arr);
  EXEC SQL set transaction read write;
  EXEC SQL insert into zipcodes (zip, city)
           values (:zipcode_rec);
  EXEC SQL commit;
}
```

2. Transactions will be covered in Section 3.9.

In this code fragment, a particular zip code value is checked to see if it is already present in the `zipcodes` table using the `EXEC SQL select into` statement. If the zip code (indicated by the host variable `:customer_rec.zip`) is not found in the `zipcodes` table, Oracle returns a positive integer in `sqlca.sqlcode`. This is checked for in the program fragment, and the user is prompted for the `City` value for this zip code and a new row is added to the `zipcodes` table.

3.4 Connecting to Oracle

The SQL `connect` statement is used to establish a connection with Oracle. Such a connection must be established before any embedded SQL statements can be executed. The following code fragment illustrates the use of the `connect` statement. The fragment, when executed, will prompt the user for the Oracle user name and password. If the connection is not established in three tries, the program exits.

```
EXEC SQL begin declare section;
  varchar userid[10], password[15];
EXEC SQL end declare section;
int     loginok=FALSE,logintries=0;

do {
  printf("Enter your USERID: ");
  scanf("%s", userid.arr);
  userid.len = strlen(userid.arr);
  printf("Enter your PASSWORD: ");
  system("stty -echo");
  scanf("%s", password.arr);
  password.len = strlen(password.arr);
  system("stty echo");
  printf("\n");
  EXEC SQL connect :userid identified by :password;
  if (sqlca.sqlcode == 0)
    loginok = TRUE;
  else
    printf("Connect Failed\n");
  logintries++;
} while ((!loginok) && (logintries <3));
if ((logintries == 3) && (!loginok)) {
    printf("Too many tries at signing on!\n");
    exit(0);
  }
```

The `userid` and `password` values cannot be provided as literal strings in the `connect` statement. They must be provided in host variables, as is shown in the above program fragment. Here, these variables are assigned values entered by the user; however, if the values were already known, these variables could be initialized as follows:

```
strcpy(userid.arr,"UUUU");
userid.len = strlen(userid.arr);
strcpy(password.arr,"PPPP");
password.len = strlen(password.arr);
```

where `UUUU` is the user name and `PPPP` is the associated password.

To disconnect from the database, which should be done at the end of the program, the following statement is used:

```
EXEC SQL commit release;
```

This commits any changes that were made to the database and releases any locks that were placed during the course of the execution of the program.

3.5 Cursors

When an embedded SQL select statement returns more than one row in its result, the simple form of the `select into` statement cannot be used anymore. To process such queries in embedded SQL, the concept of a *cursor*—a mechanism that allows the C program to access the rows of a query one at a time—is introduced. The cursor declaration associates a cursor variable with an SQL select statement. To start processing the query, the cursor must first be *opened*. It is at this time that the query is evaluated. It is important to note this fact, since the query may contain host variables that could change while the program is executing. Also, the database tables may be changing (other users are possibly updating the tables) while the program is executing. Once the cursor is opened, the `fetch` statement can be used several times to retrieve the rows of the query result, one at a time. Once the query results are all retrieved, the cursor should be `closed`.

The syntax for cursor declaration is

```
EXEC SQL declare ⟨cur-name⟩ cursor for
  ⟨select-statement⟩
  [for {read only | update [of ⟨column-list⟩]}];
```

where ⟨*cur-name*⟩ is the name of the cursor and ⟨*select-statement*⟩ is any SQL select statement associated with the cursor. The select statement, which may involve host variables, is followed by one of the following two optional clauses:

```
for read only
```

or

```
for update [of ⟨column-list⟩]
```

The `for update` clause is used in cases of *positioned* deletes or updates, discussed later in this section. The `for read only` clause is used to prohibit deletes or updates based on the cursor. If the optional `for` clause is left out, the default is `for read only`.

A cursor is opened using the following syntax:

```
EXEC SQL open ⟨cur-name⟩;
```

When the `open` statement is executed, the query associated with the cursor is evaluated, and an imaginary pointer points to the position before the first row of the query result. Any subsequent changes to the host variables used in the cursor declaration or changes to the database tables will not affect the current cursor contents.

The syntax for the `fetch` statement is

```
EXEC SQL fetch ⟨cur-name⟩ into
         ⟨host-var⟩, ..., ⟨host-var⟩;
```

where ⟨*host-var*⟩ is a host variable possibly including an indicator variable.

The syntax of the `close` cursor statement is

```
EXEC SQL close ⟨cur-name⟩;
```

The following procedure illustrates the use of a cursor to print all the rows in the `customers` table.

```
void print_customers() {
EXEC SQL declare customer_cur cursor for
         select cno, cname, street, zip, phone
         from   customers;
EXEC SQL set transaction read only;
EXEC SQL open customer_cur;
EXEC SQL fetch customer_cur into
         :customer_rec indicator :customer_rec_ind;
while (sqlca.sqlcode == 0) {
 customer_rec.cname.arr[customer_rec.cname.len] = '\0';
 customer_rec.street.arr[customer_rec.street.len] = '\0';
 printf("%6d  %10s  %20s  %6d  %15s\n",
        customer_rec.cno,customer_rec.cname.arr,
        customer_rec.street.arr,customer_rec.zip,
        customer_rec.phone);
EXEC SQL fetch customer_cur into
          :customer_rec indicator :customer_rec_ind;
}
EXEC SQL close customer_cur;
EXEC SQL commit;
}
```

Positioned Deletes and Updates

Cursors can also be used with the `delete` and `update` statements, which are referred to as positioned deletes or updates. When cursors are used for positioned deletes or updates, they must have exactly one table (the table from which the rows are to be deleted or updated) in the `from` clause of the main select statement defining the cursor. When used in this manner, the cursor declaration will have a `for update` clause, which is optionally followed by a list of columns. If such a list is provided, then only those columns that are listed there can be updated using the `update` statement. To do a positioned delete, the `for update` clause should not be followed by any list of columns. The following example illustrates a positioned delete.

```
EXEC SQL declare del_cur cursor for
    select *
    from   employees
    where  not exists
             (select 'a'
              from   orders
              where  orders.eno = employees.eno)
    for update;

  EXEC SQL set transaction read write;
  EXEC SQL open del_cur;
EXEC SQL fetch del_cur into :employee_rec;
while (sqlca.sqlcode == 0) {
  EXEC SQL delete from employees
           where current of del_cur;
  EXEC SQL fetch del_cur into :employee_rec;
}
EXEC SQL commit release;
```

This program fragment deletes all employees who do not have any orders. The cursor is defined using the `for update` clause and involves only the `employees` table in its `from` clause. Positioned updates are done in a similar manner.

3.6 Mail-Order Database Application

An application program, written and compiled using Oracle's Pro*C, is presented in this section. This program allows the user to interact with the mail-order database and provides the following functionality:

- *Add customer:* The user is prompted for the name, street address, phone, and zip code for the new customer. The customer number is generated internally using the sequence `custseq`. To maintain the referential integrity constraint (zip must also exist in the `zipcodes` table), the zip code is checked against the `zipcodes` table. If not present, the user is prompted for the city corresponding to the zip code and an entry is made in the `zipcodes` table before the new customer is added. The `insert` statement is used in this function.
- *Print customers:* This function simply prints all the customers present in the database. A simple cursor is used to accomplish this task.
- *Update customer:* The user is given the opportunity to update the street address, zip code, and phone number for a given customer. If the user updates the zip code, a similar check as in the *Add Customer* function is made to maintain the referential integrity constraint. The `update` statement is used to make the update in the database.
- *Process order:* The function keeps prompting for a valid employee number until it receives one from the user. If the customer is new, the *Add Customer* function is invoked; otherwise the customer number is requested from the user. An order number is then generated internally, using the sequence `orderseq`, and the parts and quantities are requested from the user. Finally, one row corresponding to this order is made in the `orders` table and several entries that correspond to this

order are made in the `odetails` table. Notice how carefully this function was designed so that none of the primary and foreign key constraints are violated.

- *Remove customer:* Given a customer number, this function tries to remove the customer from the database. If `orders` exists for this customer, the customer is not removed. One possibility that existed in this case was to cascade this delete, by deleting all the rows in the `orders` table and the `odetails` table that correspond to this customer, and then to delete the customer. This cascading of deletes could be done by providing explicit `delete` statements in the program; it could also be done automatically by providing the `cascade delete` property when the foreign keys were defined. However, this option was not chosen here.
- *Delete old orders:* All orders having a shipped date that is five or more years before the current date are deleted. The corresponding rows in the `odetails` table are also deleted to maintain the referential integrity.
- *Print invoice:* Given an order number, this function prints the invoice for this order, which includes the customer details, employee details, and parts in the order, including their quantities and total price. This is a typical reporting function that uses cursors to get the information from several tables.

To compile this or any other embedded SQL program for Oracle under UNIX, use the following command, where `prog.pc` is the name of the file containing the embedded SQL program.

```
make -f proc.mk EXE=prog OBJS="prog.o" build
```

The file `proc.mk` comes with the Oracle distribution and can be found under the `ORACLE_HOME` directory.

The main program is included here.

```
#include <stdio.h>
#include <string.h>
#define TRUE 1
#define FALSE 0

typedef struct {
  int cno; varchar cname[31]; varchar street[31];
  int zip; char phone[13];
} customer_record;
typedef struct {
  short cno_ind,cname_ind,street_ind,zip_ind,phone_ind;
} customer_indicator_record;
```

```
typedef struct {
  int zip; varchar city[31];
} zipcode_record;
typedef struct {
  int eno; varchar ename[31]; int zip; char hdate[12];
} employee_record;
typedef struct {
  short eno_ind,ename_ind,zip_ind,hdate_ind;
} employee_indicator_record;
typedef struct {
  int ono,cno,eno; char received[12],shipped[12];
} order_record;
typedef struct {
  short ono_ind,cno_ind,eno_ind,received_ind,shipped_ind;
} order_indicator_record;

EXEC SQL include sqlca;

void print_menu();
void add_customer();
void print_customers();
void update_customer();
void process_order();
void remove_customer();
void delete_old_orders();
void print_invoice();
void prompt(char [],char []);

void main() {
  EXEC SQL begin declare section;
    varchar userid[10], password[15];
  EXEC SQL end declare section;
  char ch;
  int  done=FALSE,loginok=FALSE,logintries=0;

  do {
    prompt("Enter your USERID: ",userid.arr);
    userid.len = strlen(userid.arr);
    printf("Enter your PASSWORD: ");
    system("stty -echo");
    scanf("%s", password.arr);getchar();
    password.len = strlen(password.arr);
    system("stty echo");
    printf("\n");
    EXEC SQL connect :userid identified by :password;
    if (sqlca.sqlcode == 0)
      loginok = TRUE;
    else
      printf("Connect Failed\n");
    logintries++;
  } while ((!loginok) && (logintries <3));
  if ((logintries == 3) && (!loginok)) {
    printf("Too many tries at signing on!\n");
    exit(0);
  }

  while (done == FALSE) {
    print_menu();
    printf("Type in your option: ");
    scanf("%s",&ch); getchar();
    switch (ch) {
      case '1': add_customer(); printf("\n"); break;
      case '2': print_customers(); printf("\n"); break;
      case '3': update_customer(); printf("\n"); break;
      case '4': process_order(); printf("\n"); break;
      case '5': remove_customer(); printf("\n"); break;
      case '6': delete_old_orders(); printf("\n"); break;
      case '7': print_invoice();
                printf("\nPress RETURN to continue");
                getchar(); printf("\n"); break;
      case 'q': case 'Q': done = TRUE; break;
      default:  printf("Type in option again\n"); break;
    }
  };
  EXEC SQL commit release;
  exit(0);
}
void print_menu() {
    printf("***************************************\n");
    printf("<1> Add a new customer\n");
    printf("<2> Print all customers\n");
    printf("<3> Update customer information\n");
    printf("<4> Process a new order\n");
    printf("<5> Remove a customer\n");
    printf("<6> Delete old orders \n");
    printf("<7> Print invoice for a given order\n");
    printf("<q> Quit\n");
    printf("***************************************\n");
  }
  void prompt(char s[], char t[]) {
    char c;
    int i = 0;

    printf("%s",s);
    while ((c = getchar()) != '\n') {
      t[i] = c;
      i++;
    }
    t[i] = '\0';
  }
```

After declaring the various record structures and indicator record structures for database access, including the `sqlca` structure, and declaring the function prototypes, the `main` function is defined. The `main` function first connects to the database and then presents a menu of choices for the user to select from. It then processes the user option by calling the appropriate function. The `prompt` function is a utility function that displays a prompt and reads a string variable. It is used throughout the application program.

3.6.1 Customer Functions

The customer-related functions are presented here. These functions allow the user to add a new customer, update the information for an existing customer, and delete an existing customer from the database. The function to print the list of customers was presented in Section 3.5.

The following is the `add_customer` function.

```
void add_customer() {
  EXEC SQL begin declare section;
    customer_record crec;
    zipcode_record  zrec;
  EXEC SQL end declare section;

  prompt("Customer Name: ",crec.cname.arr);
  crec.cname.len = strlen(crec.cname.arr);
  prompt("Street       : ",crec.street.arr);
  crec.street.len = strlen(crec.street.arr);
  printf("Zip Code     : ");
  scanf("%d",&crec.zip); getchar();
  prompt("Phone Number : ",crec.phone);

  EXEC SQL select zip, city
           into   :zrec
           from   zipcodes
           where  zip = :crec.zip;
  if (sqlca.sqlcode > 0) {
     zrec.zip = crec.zip;
     prompt("Zip not present; Enter City: ",zrec.city.arr);
     zrec.city.len = strlen(zrec.city.arr);
     EXEC SQL set transaction read write;
     EXEC SQL insert into zipcodes (zip, city)
              values (:zrec);
     EXEC SQL commit;
  }

  EXEC SQL set transaction read write;
  EXEC SQL insert into customers values
    (custseq.nextval,:crec.cname,:crec.street,
     :crec.zip,:crec.phone);
  if (sqlca.sqlcode < 0) {
    printf("\n\nCUSTOMER (%s) DID NOT GET ADDED\n",
           crec.cname.arr);
    EXEC SQL rollback work;
    return;
  }
  EXEC SQL commit;
}
```

This function requests information for a new customer and inserts the customer into the database. If the zip code value is not present in the database, this function also requests the city information and makes an entry into the `zipcodes` table.

The `update_customer` function is shown below:

```
void update_customer() {
  EXEC SQL begin declare section;
    customer_record crec;
    zipcode_record zrec;
    int cnum;
    varchar st[31];
    char ph[13], zzip[6];
  EXEC SQL end declare section;

  printf("Customer Number to be Updated: ");
  scanf("%d",&cnum);getchar();

  EXEC SQL select *
           into   :crec
           from   customers
           where  cno = :cnum;
  if (sqlca.sqlcode > 0) {
    printf("Customer (%d) does not exist\n",cnum);
    return;
  }
  crec.street.arr[crec.street.len] = '\0';
  printf("Current Street Value          : %s\n",crec.street.arr);
  prompt("New Street (n<ENTER> for same): ",st.arr);
  if (strlen(st.arr) > 1) {
    strcpy(crec.street.arr,st.arr);
    crec.street.len = strlen(crec.street.arr);
  }
  printf("Current ZIP Value      : %d\n",crec.zip);
  prompt("New ZIP (n<ENTER> for same): ",zzip);
  if (strlen(zzip) > 1) {
    crec.zip = atoi(zzip);
    EXEC SQL select zip, city
             into   :zrec
             from   zipcodes
             where  zip = :crec.zip;
```

```
      if (sqlca.sqlcode > 0) {
        zrec.zip = crec.zip;
        prompt("Zip not present; Enter City: ",zrec.city.arr);
        zrec.city.len = strlen(zrec.city.arr);
        EXEC SQL set transaction read write;
        EXEC SQL insert into zipcodes (zip, city)
           values (:zrec);
        EXEC SQL commit;
      }
    }
    printf("Current Phone Value: %s\n",crec.phone);
    prompt("New Phone (n<ENTER> for same): ",ph);
    if (strlen(ph) > 1) {
      strcpy(crec.phone,ph);
    }

    EXEC SQL set transaction read write;
    EXEC SQL update customers
             set street = :crec.street,
                 zip    = :crec.zip,
                 phone  = :crec.phone
             where cno = :crec.cno;
    if (sqlca.sqlcode < 0) {
       printf("\n\nError on Update\n");
       EXEC SQL rollback work;
       return;
    }
    EXEC SQL commit;
    printf("\nCustomer (%d) updated.\n",crec.cno);
  }
```

This function first prompts the user for the customer number. After checking if the customer exists in the database, it displays the street address, zip, and phone, and it then prompts the user for new values. The user may enter `n` followed by the `enter` key if no change is needed for a particular value. If a new zip value is entered, the city value is prompted in case the new zip is not present in the database, to ensure the integrity of the database. Finally, an update is made to the customer row in the `customers` table.

The `remove_customer` function is presented below:

```
void remove_customer() {
  EXEC SQL begin declare section;
    customer_record crec;
    int cnum,onum;
  EXEC SQL end declare section;

  printf("Customer Number to be deleted: ");
  scanf("%d",&cnum); getchar();

  EXEC SQL select *
           into :crec
           from customers
           where cno = :cnum;
  if (sqlca.sqlcode > 0) {
    printf("Customer (%d) does not exist\n",cnum);
    return;
  }

  EXEC SQL declare del_cur cursor for
      select ono from orders where cno = :cnum;

  EXEC SQL set transaction read only;
  EXEC SQL open del_cur;
  EXEC SQL fetch del_cur into :onum;
  if (sqlca.sqlcode == 0) {
      printf("Orders exist - cannot delete\n");
      EXEC SQL commit;
      return;
  }
  EXEC SQL commit;

  EXEC SQL set transaction read write;
  EXEC SQL delete from customers
           where cno = :cnum;
  printf("\nCustomer (%d) DELETED\n",cnum);
  EXEC SQL commit;
}
```

This function first prompts for the customer number. After checking to see if the customer exists in the database, it checks to see if any orders exist for this customer.

If orders exist, the delete is aborted; otherwise the customer is deleted from the database.

3.6.2 Process Orders

The `process_order` function is shown below:

```
void process_order() {
  EXEC SQL begin declare section;
    customer_record crec;
    int eenum,cnum,pnum,qqty,ord_lev,qqoh;
  EXEC SQL end declare section;
  FILE *f1; char ch; int nparts;

  EXEC SQL set transaction read only;
  do {
    printf("Employee Number: ");
    scanf("%d",&eenum); getchar();
    EXEC SQL select eno
             into   :eenum
             from   employees
             where  eno = :eenum;
    if (sqlca.sqlcode > 0)
      printf("Employee (%d) does not exist\n",eenum);
  } while (sqlca.sqlcode!=0);
  EXEC SQL commit;

  do {
    printf("New Customer (y or n)? ");
    scanf("%s",&ch); getchar();
  } while ((ch != 'y') && (ch != 'Y') &&
           (ch != 'n') && (ch != 'N'));
  if ((ch == 'y') || (ch == 'Y')) {
    add_customer();
    EXEC SQL set transaction read only;
    EXEC SQL select custseq.currval
             into   :cnum
             from   dual;
    EXEC SQL commit;
  }
```

```
else {
  printf("Customer Number: ");
  scanf("%d",&cnum); getchar();
}

EXEC SQL set transaction read only;
EXEC SQL select *
         into  :crec
         from  customers
         where cno = :cnum;
if (sqlca.sqlcode > 0){
  printf("Customer (%d) does not exist\n",cnum);
  EXEC SQL commit;
  return;
}
EXEC SQL commit;

EXEC SQL set transaction read write;
EXEC SQL insert into orders (ono,cno,eno,received)
         values (ordseq.nextval,:cnum,:eenum,sysdate);
if (sqlca.sqlcode != 0) {
  printf("Error while entering order\n");
  EXEC SQL rollback work;
  return;
}
nparts = 0;
do {
  printf("Enter pno and quantity,(0,0)to quit:  ");
  scanf("%d%d",&pnum,&qqty); getchar();
  if (pnum != 0) {
    EXEC SQL select qoh,olevel
             into   :qqoh,:ord_lev
             from   parts
             where  pno=:pnum;
    if (qqoh > qqty) {
      EXEC SQL insert into odetails
               values (ordseq.currval,:pnum,:qqty);
      if (sqlca.sqlcode == 0) {
        nparts++;
        EXEC SQL update parts
```

```
                  set qoh = (qoh - :qqty)
                  where pno=:pnum;
             if (qqoh < ord_lev){
               EXEC SQL update parts
                        set qoh = 5*olevel
                        where pno=:pnum;
               f1 = fopen("restock.dat","a");
               fprintf(f1,"Replenish part (%d) by (%d)\n",
                           pnum, 5*ord_lev - qqoh);
               fclose(f1);
             }
           }
           else  printf("Cannot add part (%d) to order\n",pnum);
          }
          else
           printf("Not enough quantity in stock for (%d)\n",pnum);
         }
       } while(pnum > 0);
       if (nparts > 0)
         EXEC SQL commit;
       else
         EXEC SQL rollback work;
       printf("NEW ORDER PROCESSING COMPLETE\n");
     }
```

This function first requests that the user enter a valid employee number, and then it asks if the customer is new or old. If the customer is new, the function invokes the routine to add a new customer; otherwise it requests the customer number. After verifying that the customer is valid, it makes an entry into the `orders` table and then repeatedly requests for the part number and quantity of each part being ordered. Each of these parts is then entered into the `odetails` table. The function terminates when the user enters 0 for the part number. If the number of parts added to the order is more than 0, the transaction is committed; otherwise the transaction is rolled back and the order entry is erased from the `orders` table. Although this function does take user input while a transaction is alive, it is not a good idea in general, as the database tables may be locked for an indefinite time and other users would not get access to these tables. One way to fix this problem is to read the parts to be ordered into an array and perform the database inserts after the user is done with the input.

3.6.3 Delete Old Orders

The `delete_old_orders` function is shown below:

```
void delete_old_orders() {
  EXEC SQL set transaction read write;
  EXEC SQL delete from ospecs
           where ono in
                  (select ono
                   from orders
                   where shipped <  (sysdate - 5*365));
  EXEC SQL delete from orders
           where shipped < (sysdate - 5*365);
  EXEC SQL commit;
  printf("ORDERS SHIPPED 5 YEARS or EARLIER DELETED!\n");
}
```

This function deletes orders that have been shipped more than five years ago. To maintain referential integrity, it also deletes the corresponding rows in the `odetails` table.

3.6.4 Print Order Invoice

The `print_invoice` function is shown below:

```
void print_invoice() {
  EXEC SQL begin declare section;
    int zzip,cnum,eenum,onum,pnum,qqty;
    varchar st[31],eename[31],ccname[31],
            ccity[31],ppname[31];
    char ph[13];
    float sum,pprice;
    order_record orec;
    order_indicator_record orecind;
  EXEC SQL end declare section;

  EXEC SQL declare od_cur cursor for
    select parts.pno, pname, qty, price
    from   odetails, parts
    where  odetails.ono = :onum and
           odetails.pno = parts.pno;
```

```
printf("Order Number: ");
scanf("%d",&onum); getchar();

EXEC SQL set transaction read only;
EXEC SQL select *
         into   :orec indicator :orecind
         from   orders
         where  ono = :onum;
if (sqlca.sqlcode == 0) {
  EXEC SQL select cno,cname,street,city,
                  customers.zip, phone
           into   :cnum,:ccname,:st,:ccity,:zzip,:ph
           from   customers, zipcodes
           where  cno = :orec.cno and
                  customers.zip = zipcodes.zip;
  ccname.arr[ccname.len] = '\0';
  st.arr[st.len] = '\0';
  ccity.arr[ccity.len] = '\0';
  printf("***************************************");
  printf("***********************\n");
  printf("Customer: %s \t Customer Number: %d \n",
         ccname.arr, cnum);
  printf("Street  : %s \n", st.arr);
  printf("City    : %s \n", ccity.arr);
  printf("ZIP     : %d \n",zzip);
  printf("Phone   : %s \n", ph);
  printf("----------------------------------------");
  printf("----------------------\n");

  EXEC SQL select eno, ename
           into   :eenum, :eename
           from   employees
           where  eno = :orec.eno;
  eename.arr[eename.len] = '\0';
  printf("Order No: %d \n",orec.ono);
  printf("Taken By: %s (%d)\n",eename.arr, eenum);
  printf("Received On: %s\n",orec.received);
  printf("Shipped  On: %s\n\n",orec.shipped);
```

```
    EXEC SQL open od_cur;
    EXEC SQL fetch od_cur
             into :pnum, :ppname, :qqty, :pprice;
    printf("Part No.              ");
    printf("Part Name    Quan.   Price     Ext\n");
    printf("------------------------------------");
    printf("-----------------------\n");
    sum = 0.0;
    while (sqlca.sqlcode == 0) {
      ppname.arr[ppname.len] = '\0';
      printf("%8d%25s%7d%10.2f%10.2f\n",pnum,
        ppname.arr, qqty, pprice, qqty*pprice);
      sum = sum + (qqty*pprice);
      EXEC SQL fetch od_cur
               into :pnum, :ppname, :qqty, :pprice;
     }
     EXEC SQL close od_cur;
     printf("------------------------------------");
     printf("-------------------------\n");
     printf("                                 ");
     printf("TOTAL:           %10.2f\n",sum);
     printf("************************************");
     printf("*************************\n");
     EXEC SQL commit;
  }
}
```

Given an order number, this function prints an invoice for the order. The invoice includes information about the customer, employee who took the order, details of the order, prices, and total due.

3.7 Recursive Queries

SQL is a powerful language to express ad hoc queries. But it has its limitations. It is impossible to express arbitrary recursive queries in SQL. However, the power of embedded SQL, which gives access to all the features of a high-level programming

Figure 3.2 `emps` table.

emps

EID	MGRID
Smith	Jones
Blake	Jones
Brown	Smith
Green	Smith
White	Brown
Adams	White

language, can be used to solve recursive queries. Consider a simple relational table `emps` defined as follows:

```
create table emps (
  eid   integer,
  mgrid integer);
```

This table has two columns: (1) `eid`, the employee ID, and (2) `mgrid`, the ID of the employee's manager. A possible instance is shown in Figure 3.2. Consider the following deductive rules that define a recursive query on the `emps` table:

```
query(X) :- emps(X,'Jones');
query(X) :- emps(X,Y), query(Y).
```

The query gets all the employees who work under `Jones` at all levels. The first deductive rule gets the employees who work directly under `Jones`. The second deductive rule is a recursive rule stating that if `X` works under `Y` and `Y` is already in the answer to the query, then `X` must also be in the answer to the query.

To solve this query using embedded SQL, the following algorithm is used:

```
insert into query
  select eid from emps where mgrid = 'Jones';
repeat
  insert into query
    select eid from query,emps where mgrid = a;
until (no more changes to query);
```

where `query` is a table with one column named `a`. Basically, the algorithm first computes the answers from the first deductive rule. It then repeatedly computes the answers from the second recursive rule until no more answers are generated. The program is as follows:

```
#include <stdio.h>
EXEC SQL begin declare section;
  int  eid, a;
EXEC SQL end declare section;
EXEC SQL include sqlca;
main()
{ int newrowadded;
/* Cursor for emps at next level (Initial answers) */
  EXEC SQL declare c1 cursor for
    select eid from emps where mgrid = :eid;
/* query(X) if emps(X,Y) and query(Y) */
  EXEC SQL declare c2 cursor for
    select eid from emps,query where mgrid = a;
/* Cursor to print the answers */
  EXEC SQL declare c3 cursor for select a from query;

  EXEC SQL create table query(
           a integer not null, primary key (a));

/*Get initial answers using Cursor c1*/

  printf("Type in employee id:");
  scanf("%d",&eid);

  EXEC SQL open c1;
  EXEC SQL fetch c1 into :a;
  while (sqlca.sqlcode == 0) {
    EXEC SQL insert into query values (:a);
    EXEC SQL fetch c1 into :a;
  }
  EXEC SQL close c1;
  EXEC SQL commit work;
/* repeat loop of algorithm */
   do {
     newrowadded = FALSE;
     EXEC SQL open c2;
     EXEC SQL fetch c2 into :a;
     while (sqlca.sqlcode == 0) {
       EXEC SQL insert into query values (:a);
       if (sqlca.sqlcode == 0)
```

```
          newrowadded = TRUE;
        EXEC SQL fetch c2 into :a;
      }
      EXEC SQL close c2;
    } while (newrowadded);
  EXEC SQL commit work;
/*Print results from query table*/

  printf("Answer is\n");
  EXEC SQL open c3;
  EXEC SQL fetch c3 into :a;
  while (sqlca.sqlcode == 0) {
    printf("%d\n",a);
    EXEC SQL fetch c3 into :a;
  }
  EXEC SQL close c3;
  EXEC SQL commit work;

  EXEC SQL drop table query;
  EXEC SQL commit work;
}/*end of main*/
```

Note the use of the `primary key` clause in the `query` table definition. Checking to see if any new row was added in the `do-while` loop in the program is based on the fact that the only column of `query` is defined as the primary key, and if a new row was indeed added, the value of `sqlca.sqlcode` would be 0.

3.8 Error Handling

Oracle reports any errors or warnings caused by the embedded SQL statements in the SQL communications area introduced in Section 3.3. There are at least two methods of handling these errors and warnings in the embedded program: *explicit handling* by writing code that inspects the `sqlca` structure and takes appropriate action, and *implicit handling* by using the `whenever` statement. Both methods are discussed in this section.

Figure 3.3 The `sqlca` structure in Oracle.

```
struct sqlca {
  char    sqlcaid[8];
  long    sqlabc;
  long    sqlcode;
  struct {
    unsigned short sqlerrml;
    char           sqlerrmc[70];
  } sqlerrm;
  char    sqlerrp[8];
  long    sqlerrd[6];
  char    sqlwarn[8];
  char    sqlext[8];
};
```

3.8.1 Explicit Handling

After each database call, errors or warnings are checked and, if necessary, are processed at that point by the application program. The SQL communications area is used for this purpose. Earlier in this chapter the `sqlcode` field of the `sqlca` structure was introduced. Now other important fields of the `sqlca` structure will be discussed. The `sqlca` structure as defined in Oracle is shown in Figure 3.3. The individual fields of the `sqlca` structure are as follows:

- `sqlcaid`: A character string that identifies the SQL communications area. It is equal to `'SQLCA   '`.
- `sqlabc`: The size of the `sqlca` structure; it is initialized to

  ```
  sizeof(struct sqlca).
  ```
- `sqlcode`: The status of the most recent SQL statement execution. A 0 value indicates successful execution of the most recent embedded SQL statement; a negative value indicates an error in the execution of the most recent embedded SQL statement; a positive value indicates a warning situation encountered while executing the most recent embedded SQL statement. A warning occurs when no data are found as a result of a `select` or `fetch` statement or when no rows are inserted as a result of an `insert` statement.
- `sqlerrm`: A substructure that contains the error message. This field should be accessed only if `sqlcode` is negative. This substructure contains two fields:

sqlerrml, the length of the error message, and sqlerrmc, the error message itself. This error message is limited to 70 characters and is not null terminated. Therefore, it should be null terminated as follows before it is used in the C program:

```
sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
```

If the error message is longer than 70 characters, a call to the sqlglm function can be made to print the entire error message as follows:

```
char error_message[512];
long buffer_length, message_length;

buffer_length = sizeof (error_message);
sqlglm(error_message,&buffer_length,&message_length);
printf("%.*s\n", message_length, error_message);
```

Note that error_message is the buffer in which Oracle would store the entire error message, buffer_length is the maximum size of the buffer (usually set to 512), and message_length is the actual length of the error message.

- sqlerrp: A character string field unused at this time.
- sqlerrd: An array of six integers used to record error diagnostic information. Only sqlerrd[2] and sqlerrd[4] are in use at this time. sqlerrd[2] records the number of rows processed successfully by a select, insert, update, or delete statement. For cursors, sqlerrd[2] is assigned 0 when the cursor is opened and incremented after each fetch. sqlerrd[4] contains the parse error offset, the position in a dynamic SQL statement that is syntactically incorrect. sqlerrd[4] is useful only in the context of dynamic SQL statements, which are discussed in Section 3.10.
- sqlwarn: An array of eight characters used as warning flags having the following meanings:
  - sqlca.sqlwarn[0] is set to W if one of the other flags is set.
  - sqlca.sqlwarn[1] is assigned a nonzero value if a character data value is truncated when assigned to a host variable.
  - sqlca.sqlwarn[3] is assigned a nonzero value if in a fetch or a select into statement the number of columns is not equal to the number of host variables in the into clause. The value assigned is the smaller of these two numbers.

  - `sqlca.sqlwarn[4]` is assigned a nonzero value if all the rows of a table are affected by a `delete` or an `update` statement.
  - `sqlca.sqlwarn[5]` is assigned a nonzero value when the compilation of a PL/SQL statement fails.

  The remaining entries in this array are unused at this time and are initialized to 0.

- `sqlext`: A character string field not in use at this time.

Based on the information provided in the `sqlca` structure, the programmer can inspect these values and take appropriate action in the embedded SQL program to handle errors. The `sqlca` structure can be initialized by using the following `define` statement:

```
#define SQLCA_INIT
```

3.8.2 Implicit Handling

One of the disadvantages of explicit handling of errors is that the code becomes cluttered with error-handling statements. To avoid this, the errors can be handled implicitly using the `whenever` statement, which is actually a directive to the Pro*C preprocessor to insert certain statements after each database call in the program under the scope of the `whenever` statement. The syntax of the `whenever` statement is

```
EXEC SQL whenever ⟨condition⟩ ⟨action⟩;
```

where ⟨*condition*⟩ is one of the following:

```
sqlerror    : an error situation  (sqlcode < 0)
sqlwarning  : a warning situation (sqlcode > 0)
not found   : data not found      (sqlcode = 1403)
```

and ⟨*action*⟩ is one of the following:

```
continue     : ignore condition and continue
do function  : call a function
do break     : perform a break out of a control
               structure
do return    : perform a return statement
goto label   : branch to a label
stop         : stop program and roll back uncommitted
               changes
```

The `whenever` statement can be used any number of times in the program. However, you must be careful, since the scope of the statement is positional, not logical. The positional nature of the scope of the `whenever` statement means that it is applicable to the statements that physically follow it until another `whenever` statement for the same ⟨*condition*⟩ is encountered. A typical use of the `whenever` statement is

```
EXEC SQL whenever sqlerror do
  sql_error("ORACLE error--\n");
```

This is placed at the beginning of the source file, and an error-handling procedure called `sql_error` is defined as follows:

```
void sql_error(char msg[]) {
  char error_message[512];
  long buffer_length, message_length;

  EXEC SQL whenever sqlerror continue;
  buffer_length = sizeof (error_message);
  sqlglm(error_message,&buffer_length,&message_length);
  printf("%.*s\n", message_length, error_message);
  EXEC SQL rollback release;
  exit(1);
}
```

This is a procedure that prints the error message using the `sqlglm` procedure call and exits the program. Notice the use of the

```
EXEC SQL whenever sqlerror continue;
```

statement in this procedure. This ensures that an infinite loop is not created if the `rollback` statement or any other database call in this procedure encounters an error.

3.9 Transaction Control

A transaction is a sequence of database statements that must execute as a whole to maintain consistency. If for some reason one of the statements fails in the transaction, all the changes caused by the statements in the transaction should be undone to maintain a consistent database state. Oracle provides the following statements to create transactions within the embedded SQL program.

- `EXEC SQL set transaction read only`. This is typically used before a sequence of `select` or `fetch` statements that do only a read from the database.

The statement begins a read-only transaction. At the end of the sequence of statements, the `commit` statement is executed to end the transaction.

- `EXEC SQL set transaction read write`. This is typically used before a sequence of statements that performs some updates to the database. This statement begins a read-write transaction. The transaction is terminated by either a `commit` statement that makes the changes permanent in the database or a `rollback` statement that undoes all the changes made by all the statements within the transaction.
- `EXEC SQL commit`. This statement commits all the changes made in the transaction and releases any locks placed on tables so that other processes may access them.
- `EXEC SQL rollback`. This statement undoes all the changes made in the course of the transaction and releases all locks placed on the tables.

Notice the use of these statements in the mail-order database application program discussed earlier. An important point to note while creating transactions in the embedded SQL program is that while a transaction is active, user input should be avoided, if at all possible. This is because the tables are locked by the system during the transaction, and other processes may not get access to the database tables if these locks are not released soon. By waiting for user input during a transaction, there is the danger of a long delay before the input is received by the program, which means that the transaction could be kept active for long durations.

3.10 Dynamic SQL

The embedded SQL statements seen so far have been static in nature—i.e., they are fixed at compile time. However, there are situations when the SQL statements to be executed in an application program are not known at compile time. These statements are built spontaneously as the program is executing. These are referred to as *dynamic* SQL statements, which can be executed through several mechanisms, discussed in this section.

3.10.1 The `execute immediate` Statement

The simplest form of executing a dynamic SQL statement is to use the `execute immediate` statement. The syntax of the `execute immediate` statement is

```
EXEC SQL execute immediate ⟨host-var⟩;
```

where ⟨*host-var*⟩ is a host variable defined as a character string and has as its value a valid SQL statement, which cannot have any built-in host variables. It must be a complete SQL statement ready to be executed. Furthermore, it is restricted to be one of the following: `create table`, `alter table`, `drop table`, `insert`, `delete`, or `update`. Dynamic select statements are not allowed. Another approach is necessary to perform dynamic selects and is discussed later in this section. An example of the `execute immediate` statement is shown below:

```
#include <stdio.h>
EXEC SQL begin declare section;
  char sql_stmt[256];
  varchar userid[10], password[15];
EXEC SQL end declare section;
EXEC SQL include sqlca;

void main() {
 strcpy(username.arr,"book");
 username.len = strlen(username.arr);
 strcpy(password.arr,"book");
 password.len = strlen(password.arr);
 EXEC SQL connect :username identified by :password;

 strcpy(sql_stmt,
   "update employees set hdate=sysdate where eno = 1001");

 EXEC SQL set transaction read write;
 EXEC SQL execute immediate :sql_stmt;
 EXEC SQL commit release;
 exit(0);
}
```

In the above program, `sql_stmt` is the host variable that has as its value a valid SQL `update` statement. When executed successfully, it sets the `hired date` value for employee `1001` to the current date.

3.10.2 The `prepare` and `execute using` Statements

One of the drawbacks of the `execute immediate` statement is that the SQL statement that it executes has to be compiled each time the `execute immediate` statement is executed. This can cause a lot of overhead if the SQL statement has to be executed many times in the program. Another problem is that the SQL statement cannot have host variables, which reduces the flexibility of dynamic statements. To address these problems, the `prepare` and `execute using` statements are introduced. The dynamic SQL statement that needs to be executed is compiled once using the `prepare` statement and then executed any number of times using the `execute using` statement. This approach also allows host variables to be present in the dynamic SQL statement. The syntax of the `prepare` statement is

```
EXEC SQL prepare s from :sql_stmt;
```

where `sql_stmt` is a host variable defined as a character string and has as its value a valid SQL statement, which may include host variables. The `prepare` statement basically compiles the dynamic SQL statement and stores the compiled form in the variable `s`, which need not be declared. The syntax of the `execute using` statement is

```
EXEC SQL execute s using :var1, ..., :varn;
```

where `s` is a previously prepared compiled form of an SQL statement, and `var1, ..., varn` are host variables that will substitute the corresponding host variables in the dynamic SQL statement (assuming there are `n` host variables used in the dynamic SQL statement). An example of this approach is shown below:

```
#include <stdio.h>
EXEC SQL begin declare section;
  char sql_stmt[256];
  int num;
  varchar userid[10], password[15];
EXEC SQL end declare section;
EXEC SQL include sqlca;

void main() {

  strcpy(username.arr,"book");
  username.len = strlen(username.arr);
  strcpy(password.arr,"book");
  password.len = strlen(password.arr);
  EXEC SQL connect :username identified by :password;

  strcpy(sql_stmt,
   "update employees set hdate=sysdate where eno = :n");

  EXEC SQL set transaction read write;
```

```
    EXEC SQL prepare s from :sql_stmt;

    do {
      printf("Enter eno to update (0 to stop):>");
      scanf("%d",&num);
      if (num > 0) {
        EXEC SQL execute s using :num;
        EXEC SQL commit;
      }
    } while (num > 0);

    EXEC SQL commit release;
    exit(0);
  }
```

In the above program, the user is prompted for the employee number several times, and the particular employee's hired date is updated using the employee number entered by the user. Notice that the `prepare` statement is used once and the `execute using` statement is used in a loop several times.

3.10.3 Dynamic Select

The previous approaches discussed cannot be used to do dynamic selects because the number of columns in the `select` clause of the select statement and their data types are unknown at compile time. It is not possible to use the `into` clause in the `select` or the `fetch` statements, as the number of host variables to be used is unknown. To solve this problem, a new data structure is introduced, called the `SQL Descriptor Area`, or `sqlda`. This data structure will store the results of the current `fetch`, and the embedded program can obtain the current row from the `sqlda`.

The `sqlda` structure is shown in Figure 3.4. The individual fields of the `sqlda` structure are explained with comments next to their definition. Some of these fields (such as `N`, `M`, and `Y`) of the `sqlda` structure are initialized when initial space for the structure is allocated using the `sqlald` function. Other fields (such as `T`, `F`, `S`, `C`, `X`, and `Z`) are assigned values when the `describe` statement is executed. The actual values (fields `V`, `L`, and `I`) of the columns being retrieved are assigned values when the `fetch` statement is executed.

The `describe` statement returns the names, data types, lengths (including precision and scale), and `null/not null` statuses of all the columns in a compiled form of a dynamic SQL select statement. It must be used after the dynamic SQL

Figure 3.4 The `sqlda` structure in Oracle.

```
struct sqlda {
  long       N; /* Maximum number of columns
                   handled by this sqlda       */

  char     **V; /* Pointer to array of pointers
                   to column values            */
  long      *L; /* Pointer to array of lengths
                   of column values            */
  short     *T; /* Pointer to array of data
                   types of columns            */
  short    **I; /* Pointer to array of pointers
                   to indicator values      */

  long       F; /* Actual Number of columns found
                   by describe                 */

  char     **S; /* Pointer to array of pointers
                   to column names             */
  short     *M; /* Pointer to array of max lengths
                   of column names             */
  short     *C; /* Pointer to array of actual
                   lengths of column names     */

  char     **X; /* Pointer to array of pointers
                   to indicator variable names */
  short     *Y; /* Pointer to array of max lengths
                   of indicator variable names  */
  short     *Z; /* Pointer to array of actual lengths
                   of indicator variable names */
};
```

statement has been compiled using the `prepare` statement. The syntax of the `describe` statement is

```
   EXEC SQL describe select list for s into da;
```

where `s` is the compiled form of the dynamic SQL statement and `da` is the `sqlda` structure.

The cursor manipulation for dynamic selects using the `sqlda` is done as follows:

```
EXEC SQL declare c cursor for s;

EXEC SQL open  c using descriptor da;
EXEC SQL fetch c using descriptor da;
.
.
.
EXEC SQL fetch c using descriptor da;
EXEC SQL close c;
```

Notice the `USING DESCRIPTOR` phrase used in the `open` and `fetch` statements. Also notice the cursor declaration that specifies the compiled form `s` of the dynamic select. These statements should follow the `prepare` statement.

An embedded SQL program that involves a dynamic `select` is shown below. The program has a string variable assigned to the following select statement:

```
select eno,ename,hdate
from   employees
where  eno>=1;

/* Dynamic Select Program */

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

#define MAX_ITEMS         40/* max number of columns*/
#define MAX_VNAME_LEN     30/* max length for column names*/
#define MAX_INAME_LEN     30/* max length of indicator names*/

EXEC SQL begin declare section;
  varchar username[20];
  varchar password[20];
  char    stmt[256];
EXEC SQL end declare section;

EXEC SQL include sqlca;
EXEC SQL include sqlda;

SQLDA *da;

extern SQLDA *sqlald();
extern void sqlnul();
int process_select_list();

main() {
  int i;

  /* Connect to the database. */
  strcpy(username.arr,"book");
  username.len = strlen(username.arr);
  strcpy(password.arr,"book");
  password.len = strlen(password.arr);
  EXEC SQL connect :username identified by :password;

  /* Allocate memory for the SQLDA da and pointers
     to indicator variables and data. */
  da = sqlald (MAX_ITEMS, MAX_VNAME_LEN, MAX_INAME_LEN);
  for (i = 0; i < MAX_ITEMS; i++) {
      da->I[i] = (short *) malloc(sizeof(short));
      da->V[i] = (char *) malloc(1);
  }

  strcpy(stmt,
  "select eno,ename,hdate from employees where eno>=1");

  EXEC SQL prepare s from :stmt;

  process_select();

  /* Free space */
  for (i = 0; i < MAX_ITEMS; i++) {
      if (da->V[i] != (char *) 0)
         free(da->V[i]);
      free(da->I[i]);
  }
  sqlclu(da);

  EXEC SQL commit work release;
  exit(0);
}
```

```
void process_select(void) {
  int i, null_ok, precision, scale;
  EXEC SQL declare c cursor for s;


  EXEC SQL open c using descriptor da;

  /* The describe function returns their names, datatypes,
     lengths (including precision and scale), and
     null/not null statuses. */
  EXEC SQL describe select list for s into da;

  /* Set the maximum number of array elements in the
     descriptor to the number found. */
  da->N = da->F;

  /* Allocate storage for each column.  */
  for (i = 0; i < da->F; i++) {
    /* Turn off high-order bit of datatype */
    sqlnul (&(da->T[i]), &(da->T[i]), &null_ok);
    switch (da->T[i]) {
      case  1 : break; /* Char data type */
      case  2 : /* Number data type */
        sqlprc (&(da->L[i]), &precision, &scale);
        if (precision == 0) precision = 40;
        if (scale > 0) da->L[i] = sizeof(float);
        else da->L[i] = sizeof(int);
        break;
      case 12 : /* DATE datatype */
        da->L[i] = 9;
        break;
    }
    /* Allocate space for the column values.
       sqlald() reserves a pointer location for
       V[i] but does not allocate the full space for
       the pointer.  */
     if (da->T[i] != 2)
       da->V[i] = (char *) realloc(da->V[i],da->L[i] + 1);
     else
       da->V[i] = (char *) realloc(da->V[i],da->L[i]);
  /* Print column headings, right-justifying number
         column headings. */
  if (da->T[i] == 2)
     if (scale > 0)
       printf ("%.*s ", da->L[i]+3, da->S[i]);
     else
       printf ("%.*s ", da->L[i], da->S[i]);
  else
      printf ("%-.*s ", da->L[i], da->S[i]);

  /* Coerce ALL datatypes except NUMBER to
     character. */
  if (da->T[i] != 2) da->T[i] = 1;

  /* Coerce the datatypes of NUMBERs to float or
     int depending on the scale. */
  if (da->T[i] == 2)
   if (scale > 0) da->T[i] = 4;  /* float */
    else da->T[i] = 3;  /* int */
}
printf ("\n\n");

/* FETCH each row selected and print the
   column values. */
EXEC SQL whenever not found goto end_select_loop;
for (;;) {
  EXEC SQL fetch c using descriptor da;
  for (i = 0; i < da->F; i++) {
    if (*da->I[i] < 0)
        if (da->T[i] == 4)
          printf ("%-*c ",(int)da->L[i]+3, ' ');
        else
          printf ("%-*c ",(int)da->L[i], ' ');
    else
        if (da->T[i] == 3)     /* int datatype */
          printf ("%*d ", (int)da->L[i],*(int *)da->V[i]);
        else if (da->T[i] == 4)     /* float datatype */
          printf ("%*.2f ", (int)da->L[i],*(float *)da->V[i]);
        else                      /* character string */
          printf ("%-*.*s ", (int)da->L[i],
                    (int)da->L[i], da->V[i]);
      }
      printf ("\n");
    }
    end_select_loop:
    EXEC SQL close c;
    return;
  }
```

The following characteristics should be noted about the program:

- The `sqlda` structure needs to be initialized by the function call

  ```
  da = sqlald (MAX_ITEMS, MAX_VNAME_LEN,
               MAX_INAME_LEN);
  ```

  At this point, `N`, the maximum number of columns; `M`, the maximum size of column names; and `Y`, the maximum size of indicator variable names, are initialized.

- Immediately after the `sqlda` structure is initialized, the space for the indicator and column value pointers (`I` and `V` fields) must be allocated as follows.

  ```
  for (i = 0; i < MAX_ITEMS; i++) {
      da->I[i] = (short *) malloc(sizeof(short));
      da->V[i] = (char *) malloc(1);
  }
  ```

- Before the dynamic statement can be used, it should be compiled using the following statement:

  ```
  EXEC SQL prepare s from :stmt;
  ```

- The `T` field has encoded in it the `null/not null` status of the column in its high-order bit. To turn it off, use the following procedure call:

  ```
  /* Turn off high-order bit of datatype */
  sqlnul (&(da->T[i]), &(da->T[i]), &null_ok);
  ```

- After determining the data types (`T` field) of each column, the `L` field must be set to indicate the maximum lengths of each column value. After this is done, the actual space to store the column values must be allocated, which is based on the `L` field.

- The three data types, `Character String (T = 1)`, `Number (T = 2)`, and `Date (T = 12)`, are handled in this program.

- The `sqlprc()` function call is used to extract precision and scale from the length `(da->L[i])` of the `NUMBER` columns.
- The column names are obtained from the `S` field and are available after the `describe` statement has executed.
- The column values are obtained from the `V` field after the `fetch` takes place.
- At the end, it is good practice to free the space occupied by the `sqlda` structure. This is done by the following:

```
/* Free space */
for (i = 0; i < MAX_ITEMS; i++) {
    if (da->V[i] != (char *) 0)
       free(da->V[i]);
    free(da->I[i]);
}
sqlclu(da);
```

3.11 Pro*C++

Oracle's Pro*C/C++ allows the programmer to embed SQL statements in a C++ program. In this section, three sample programs written in C++ are presented. These sample programs illustrate various aspects of embedded-SQL programming in C++ such as single answer querying using the `select into` statement, multiple answer querying using cursors, and dynamic querying using cursors and the `prepare` statement.

3.11.1 Compiling Pro*C++ programs

To compile Pro*C++ programs, use the following command:

```
make -f proc.mk EXE=prog OBJS="prog.o" cppbuild
```

where `prog.pc` is the name of the Pro*C++ program and `proc.mk` is the Oracle-supplied makefile.

3.11.2 A Simple Query Example

The following program prompts the user for a member id, then queries the member table for the member's last name, address, and email, and prints the information.

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

EXEC SQL begin declare section;
  varchar user[20];
  varchar pass[20];

  struct memberRecord {
      varchar   lname[16];
      varchar   address[50];
      varchar   email[31];
  } mrec;

  struct memberindRecord {
      short     lname_ind;
      short     addr_ind;
      short     email_ind;
  } mrec_ind;

  char mmid[8];
EXEC SQL end declare section;

class member {
  char    lname[16];
  char    address[50];
  char    email[31];
public:
  member(const memberRecord&, const memberindRecord&);
  friend ostream& operator<<(ostream&, member&);
};

member::member(const memberRecord& m, const memberindRecord& i) {
  strncpy(lname, (char *)m.lname.arr, m.lname.len);
  lname[m.lname.len] = '\0';
  if (i.addr_ind < 0)
    strncpy(address,"NULL",4);
  else
    strncpy(address, (char *)m.address.arr, m.address.len);
  address[m.address.len] = '\0';
  if (i.email_ind < 0)
    strncpy(email,"NULL",4);
  else
    strncpy(email, (char *)m.email.arr, m.email.len);
  email[m.email.len] = '\0';
}

ostream& operator<<(ostream& s, member& m) {
  return s << m.lname << " " << m.address
           << " " << m.email << endl;
}

#include <sqlca.h>

int main() {

  user.len = strlen(strcpy((char *)user.arr, "book"));
  pass.len = strlen(strcpy((char *)pass.arr, "book"));

  EXEC SQL connect :user identified by :pass;

  cout << "\nConnected to ORACLE as user: "
       << (char *)user.arr << endl << endl;

  while (1) {
    cout << "Enter member number (0 to quit): ";
    gets(mmid);
    if (strcmp(mmid,"0") == 0)
      break;
    EXEC SQL select member.lname, member.address,
                    member.email
             into   :mrec indicator :mrec_ind
             from   member
             where  member.mid = :mmid;
    if (sqlca.sqlcode == 0) {
      member m(mrec, mrec_ind);
      cout << m;
    }
    else {
      cout << "Not a valid member number." << endl;
      cout << sqlca.sqlerrm.sqlerrmc << endl;
    }
  }

  EXEC SQL commit work release;
  exit(0);
}
```

A class called `member` is declared that has three instance variables `lname`, `address`, and `email`. The information retrieved from the database is used to create a `member` object using the constructor method defined for this class. A method to print the `member` object to the output stream is defined.

The main program first connects to the database. It then repeatedly asks the user for a member id and performs the query to retrieve the necessary information from the database. This information is sent to the constructor method to create a `member` object, which is then sent to the output stream. The address and email information retrieved from the database is checked for `null` values in the constructor method using indicator variables.

An important point to note is that the database-related statements in this program are exactly the same as would have been written in a Pro*C program. This is indeed true for any Pro*C++ program.

3.11.3 Multiple Answer Query using Cursors

Consider the following two additional tables in the investment portfolio database.

```
create table analyst (
  aid       varchar2 (4),
  name      varchar2 (15)  not null,
  password  varchar2 (8),
  primary key (aid)
);
```

```
create table rating (
  aid     varchar2(4),
  symbol  varchar2(8),
  rating  number  check (rating in (1,2,3,4,5)),
  primary key (aid, symbol),
  foreign key (aid) references analyst,
  foreign key (symbol) references security
);
```

The `analyst` table records information about various analysts who rate the securities on a scale of 1–5. The ratings of securities themselves are recorded in the `ratings` table.

The following Pro*C++ program reads rows from the `analyst` and `security` tables and creates the rating table by assigning a random number between 1 and 5 for each analyst-security combination. The net effect of running this program is that every security gets a rating from each of the analysts.

```
#include <fstream.h>
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#define UNAME_LEN 20
#define PWD_LEN   40

exec sql begin declare section;
  varchar username[UNAME_LEN];
  varchar password[PWD_LEN];

  // Host variables for input
  int totalRatIns, totalAnalysts, totalSecurities;

  // Host variables for analyst id, security symbol, and rating
  varchar analystId[5];
  varchar symbol[9];
  int rating;
  // indicator variables for analyst id and security symbol
  short analystIdInd, symbolInd;
exec sql end declare section;

#include <sqlca.h>

// Error handler
void sql_error (char *msg);

void main() {
  // Register sql_error as the error handler
  exec sql whenever sqlerror
      do sql_error("ORACLE ERROR: ");

  cout << "Enter userid : ";
  cin  >> username.arr;
  cout << "Enter your password : ";
  cin  >> password.arr;

  username.len = strlen((char *) username.arr);
  password.len = strlen((char *) password.arr);

  exec sql connect :username identified BY :password;
  cout << "\nConnected to ORACLE as user: "
       << (char *)username.arr << endl;

  srand(time(NULL));
  totalRatIns = 0;
  totalAnalysts = 0;
  totalSecurities = 0;

  exec sql declare analyst_cur cursor for
           select aid
           from   analyst;

  exec sql declare security_cur cursor for
           select symbol
           from   security;
```

```
exec sql open analyst_cur;
exec sql fetch analyst_cur
         into :analystId indicator :analystIdInd;

while (sqlca.sqlcode == 0) {
  totalAnalysts++;
  exec sql open security_cur;
  exec sql fetch security_cur
           into :symbol indicator :symbolInd;
  while (sqlca.sqlcode == 0) {
    totalSecurities++;
    exec sql set transaction read write;
    rating = 1 + rand()%5;
    exec sql insert into rating values
       (:analystId,:symbol,:rating);
    if (sqlca.sqlcode != 0) {
      cout << " Error while inserting rating for "
           << analystId.arr  << " symbol "
           << symbol.arr << endl;
      exec sql rollback release;
    }
    else {
      cout << " Analyst " << analystId.arr
           << " rated " << symbol.arr
           << " as " << rating << endl;
      totalRatIns++;
      exec sql commit work;
    }
   exec sql fetch security_cur
            into :symbol INDICATOR :symbolInd;
 }
 exec sql fetch analyst_cur
          into :analystId indicator :analystIdInd;
}
exec sql close analyst_cur;
exec sql close security_cur;
cout << endl << "Total analysts = " << totalAnalysts << endl;
cout << endl << "Total securities = "
     << totalSecurities << endl;
  cout << endl << "Total ratings inserted = "
       << totalRatIns << endl << endl;

  // Disconnect from ORACLE
  exec sql commit work release;
  exit(0);
}

void sql_error(char *msg) {
  exec sql whenever sqlerror continue;
  cout << msg << sqlca.sqlerrm.sqlerrmc << endl;
  exec sql rollback release;
  exit(1);
}
```

The program begins by connecting to the database. It then uses two cursors to query the `analyst` and `security` tables. A nested loop is set up to go through every analyst-security pair. For each such pair, a random number between 1 and 5 is generated and the values are inserted into the `rating` table. Appropriate error checks are made and finally the total number of analysts, securities, and ratings is printed. Once again, notice how the database-related statements in this program are exactly the same as in a Pro*C program.

3.11.4 Dynamic SQL Query

The following program uses dynamic SQL to retrieve all the analyst ratings for a given security symbol.

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

#define UNAME_LEN 20
#define PWD_LEN   40

exec sql begin declare section;
  varchar username[UNAME_LEN];
  varchar password[PWD_LEN];
  varchar sqlstmt[80];
  varchar aname[16];
  varchar symbol[9];
  int     rating;
exec sql end declare section;

#include <sqlca.h>

// Error handler
void sql_error(char *msg);

main() {
  // Register sql_error as the error handler
  exec sql whenever sqlerror
       do sql_error("ORACLE ERROR: ");

  cout << "Enter userid : ";
  cin  >> username.arr;
  cout << "Enter your password : ";
  cin  >> password.arr;

  username.len = strlen((char *) username.arr);
  password.len = strlen((char *) password.arr);

  exec sql connect :username identified BY :password;
  cout << "\nConnected to ORACLE as user: "
       << (char *)username.arr << endl;

  strcpy((char *)sqlstmt.arr,
         "select a.name, r.rating ");
  strcat((char *)sqlstmt.arr,
         "from rating r, analyst a ");
  strcat((char *)sqlstmt.arr,
         "where r.aid = a.aid and r.symbol = :v1");
  sqlstmt.len = strlen((char *)sqlstmt.arr);

  cout << "Enter Symbol: ";
  cin >> symbol.arr;
  symbol.len = strlen((char *)symbol.arr);
```

```
    cout << (char *)sqlstmt.arr << endl;
    cout << "   v1 = " << symbol.arr << endl;

    exec sql prepare S from :sqlstmt;
    exec sql declare c cursor for S;
    exec sql open c using :symbol;
    exec sql whenever not found do break;
    while (1) {
      exec sql fetch c into :aname,:rating;
      aname.arr[aname.len] = '\0';
      cout << (char *)aname.arr << "  "
           << rating << endl;
    }

    printf("\nQuery returned %d row%s.\n\n",
           sqlca.sqlerrd[2],
           (sqlca.sqlerrd[2] == 1) ? "" : "s");

    exec sql close c;
    exec sql commit release;
    exit(0);
  }

  void sql_error(char *msg) {
    cout << endl << msg << endl;
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
    cout << sqlca.sqlerrm.sqlerrmc << endl;
    exec sql whenever sqlerror continue;
    exec sql close c;
    exec sql rollback release;
    exit(1);
  }
```

After connecting to the database, the program assigns the SQL query to a string variable. The SQL query has a place holder for the security symbol. The user is prompted for the security symbol for which the ratings are to be retrieved. The `prepare` statement is used to create a compiled form of the SQL statement. A cursor is then declared for this prepared statement. The cursor is opened by providing the actual value for the place holder in the query. The rows of the cursor are retrieved one at a time and the ratings are sent to the output stream.

Exercises

Grade Book Database Problems

3.1 Write an embedded SQL program that prompts the user for the student ID and prints a report for the student containing information about all the courses taken by the student and the average and grade obtained by the student in these courses. The format of the report is shown below:

```
Student ID:   1111
Student NAME: Nandita Rajshekhar

TERM LINENO CNO     TITLE     AVERAGE GRADE
---- ------ ------  --------  ------- -----
F96  1031   CSc481  Automata   98.50    A
.
.
.
-------------------------------------------
```

3.2 Write an embedded SQL program that prompts the user for the term and line number for a course and prints a course report containing information about the students enrolled in the course, their scores in each of the components, and their average and grade for the course. At the end of the report, the total number of A, B, C, D, and F grades awarded in the course should be listed along with the course average (the average of the individual student averages). The report should be sorted based on the student name. The format of the report is shown below:

```
      CSc 481 Automata (LineNo 1585) W98

SID  LNAME FNAME EXAM1 EXAM2 HW  QUIZ AVG   GRADE
-------------------------------------------------
2243 Green Tony  76    78    225 100  89.10   A
0213 Jones Pat   60    67    166  94  74.12   C
.
.
.
-------------------------------------------------
Total As: 5
Total Bs: 3
Total Cs: 3
Total Ds: 2
Total Fs: 1
Course Average: 78.66
```

3.3 Write an embedded SQL program that prints a report containing all courses taught along with total number of students in each course. The report should also contain the number of students enrolled in courses per calendar year and must end with a grand total of all students in all courses. The courses must be sorted according to the term and calendar year—i.e., the courses for earlier calendar years must appear before the courses for later calendar years. Within a year, the courses must be listed according to the term order: `winter`, `spring`, `summer`, `fall`. You may assume that the last two characters of the term column contains a two-digit year, and the possible values of the term column are `wXX`, `spXX`, `suXX`, and `fXX`, where `XX` is the two-digit year. The format of the report is shown below:

```
Term CNO    Course-Title   #Students Year-Total
---- ------ -------------  --------- ----------
w97  CSc226 Programming I      24
sp97 CSc227 Programming II     32
f97  CSc343 Assembly Prog      27
                                          83
sp98 CSc226 Programming I      39
f98  CSc343 Assembly Prog      22
                                          61
.
.
.
-------------------------------------------------
Total number of students:                697
```

3.4 Write an embedded SQL program that will perform the task of entering student scores for a given component in a course. The program should first prompt for the term and line number for the course. It should then list all the components of the course, request the user to select the component for which the scores are to be entered, prompt the user for the names of the students enrolled in the course, one at a time, and read in the scores for each of the students for the selected component. The program should process these scores by inserting the appropriate row in the `scores` table. The program should be robust and should handle all possible error situations, such as invalid term, invalid line number, invalid component selection, and invalid scores.

To verify that this program indeed worked, write another embedded SQL program that prompts the user for the term, line number, and component name and prints the scores for all the students for this component.

3.5 Write an embedded SQL program that will perform the task of updating student scores and dropping students from a course. The program should begin by prompting the user for the term and line number of the course. It should then display the following menu:

```
(1) Display Students
(2) Display Student Score
(3) Update Student Score
(4) Drop Student from Course
(5) Quit
```

The `Display Students` option, when selected by the user, should display the names of all the students enrolled in the course. The `Display Student Score` option, when selected by the user, should prompt the user for the student ID and component name and then display the student's score in the component. The `Update Student Score` option, when selected, should prompt the user for the student ID and component name, display the current score, and request the user to enter the new score. It should then update the database with the new score. The `Drop Student from course` option, when selected, will prompt the user for the student ID and then drop the student from the course (delete the row in the `enrolls` table). Since the `scores` table has a foreign key referring to the `enrolls` table, the corresponding rows must first be deleted from the `scores` table before the student row is deleted from the `enrolls` table. The program should be robust and should handle all possible error situations.

Mail-Order Database Problems

3.6 Write an embedded SQL program that will produce a report containing the ten most ordered parts during a calendar year. The program should prompt for a year and then produce a report with the following format:

```
                  YEAR: 1997

Rank PNO   Part-Name          Quantity-Ordered
----------------------------------------------
1.   10506 Land Before Time I       98
2.   10507 Land Before Time II      87
.
.
.
10.  10900 Dr. Zhivago              24
----------------------------------------------
```

3.7 Write an embedded SQL program that will produce a performance report for all employees. The report should contain the number and name of the employee along with the total sales generated by the employee. The report should also contain the hire date of the employee and the number of months from the hire date to the current date. In addition, the report should list the average sales per 12 months (called `RATING`) for each employee. This average can be used to compare the sales performances of the employees. The report should be sorted in decreasing order of this average performance rating and should have the following format:

```
                  PERFORMANCE REPORT
                 Dated: 12 April 1998

RANK ENO   ENAME  HIRED-ON  MONTHS TOTAL-SALES RATING
-----------------------------------------------------
1.   1000  Jones  12-DEC-95   29     2,900      1200
2.   3000  Smith  01-JAN-92   76     6,500      1027
3.   2000  Brown  01-SEP-92   68     4,000       706
-----------------------------------------------------
```

3.8 Write an embedded SQL program that will produce a mailing list of customers living in a given city. The program should prompt the user for a city and then produce the report. The format of the report is a simple listing of customers and their mailing addresses, similar to a mailing label program output.

3.9 Write an embedded SQL program that will update an incorrect zip code in the database. The program should prompt the user for the incorrect and then the correct zip code. It should then replace all the occurrences of the incorrect zip code by the correct zip code in all the tables.

To verify that the program did work, write another embedded SQL program that reads in a zip code and prints all the customers and employees living at that zip code. This program should be run before and after the previous program is executed in order to verify the execution of the previous program.

3.10 Write an embedded SQL program that reads data from a text file and updates the `qoh` column in the `parts` table. The text file consists of several lines of data, with each line containing a part number followed by a positive quantity. The `qoh` value in the `parts` table for the part should be increased by the quantity mentioned next to the part number. Assume that the last line of the text file ends with a part number of 0 and a quantity of 0. The dynamic SQL statements `PREPARE` and `EXECUTE USING` should be used to accomplish this task.

Investment Portfolio Database Problems

3.11 Write an embedded SQL program that will print the current portfolio for a member. The program should prompt the user for the member id and then print the current portfolio in the following format:

```
                         MY PORTFOLIO

Symbol Shares   Current  Market    Purchase    Gain    %Gain
                 PPS     Value       Price
-------------------------------------------------------------
ORCL   100.00    23.25   2325.00   2708.06   -383.06   -14.14
SEG    100.00    30.00   3000.00   3244.62   -244.62    -7.53
-------------------------------------------------------------
  Security Value:        5325.00   5952.68   -627.68   -10.54
  Cash Balance:         94047.33
  Account Value:        99372.33
-------------------------------------------------------------
```

3.12 Write an embedded SQL program to view the ratings of a particular security. The program should prompt the user for the security symbol and produce the ratings list as follows:

```
Symbol: ORCL
Company: Oracle Corporation
Ratings:  Strong Buy  (rating = 1) : *****
          Buy         (rating = 2) : **
          Hold        (rating = 3) : **
          Sell        (rating = 4) :
          Strong Sell (rating = 5) :
          Consensus:       1.67
```

The number of stars after each rating is the number of analysts rating the start with that particular rating. The `Consensus` is the weighted mean of the ratings.

3.13 Write an embedded SQL program that prompts the user for the month and year and produces a monthly transaction log with the following format:

```
               MONTHLY TRANSACTION REPORT
                     05/1999
-------------------------------------------------------
  Date      Type Symbol Shares PPS   Commission  Amount
-------------------------------------------------------
20-MAY-1999  buy  ORCL  100.00 26.81   26.80   2708.05
20-MAY-1999  buy  SEG   100.00 32.12   32.11   3244.61
-------------------------------------------------------
```

3.14 Write an embedded SQL program that prompts the user for a substring of a company name and prints the price quotes for all securities whose company name has the substring. For example, if the user provided `Qu` as the substring, the following quotes (in the format specified) must be generated:

```
Symbol  Company           Last Sale    Ask     Bid
----------------------------------------------------
QNTM  Quantum Corp          19.93     20.00   19.93
EAGL  Quotes                 0.00     37.87   37.75
BMY   Bristol-Myers Squibb  66.00      null    null
----------------------------------------------------
```

Recursive Query Problems

3.15 Recursive queries are easily expressed in rule-based languages such as Datalog or Prolog. Rules are generally of the form

```
P :- Q1, Q2, ..., Qn
```

and are interpreted as follows:

```
if Q1 and Q2 and ... and Qn then P
```

Consider the database table

```
parent(child,childs_parent)
```

that records information about persons and their parents. The following set of recursive rules describes certain family relationships based on the `parent` relation:

```
sibling(X,Y) :- parent(X,Z), parent(Y,Z), X <> Y.

cousin(X,Y)  :- parent(X,Xp), parent(Y,Yp),
                sibling(Xp,Yp).
cousin(X,Y)  :- parent(X,Xp), parent(Y,Yp),
                cousin(Xp,Yp).

related(X,Y) :- sibling(X,Y).
related(X,Y) :- related(X,Z), parent(Y,Z).
related(X,Y) :- related(Z,Y), parent(X,Z).
```

The rule for `sibling` says that if two different persons `X` and `Y` have the same parent `Z`, then `X` and `Y` are siblings. The other rules are interpreted in a similar manner. Write an embedded SQL program that implements the following menu-based application that queries the family relationship database.

```
     MENU
(1) Given person, find all siblings
(2) Given person, find all cousins
(3) Given person, find all related persons
(4) Given two persons, test to see if they are
    siblings, cousins or related
(5) Quit
```

3.16 Consider a database describing a network of train connections in a large metropolitan train system. Assume that there are several lines and that trains operate between stations on particular lines. A relation, called `leg`, contains data indicating which two stations are directly connected without any intermediate stops. This relation has 3 columns: `line`, `depart`, and `arrive`. A row in this table indicates a direct connection starting at `depart` station and arriving at `arrive` station without stops on the line named `line`. A second relation, called `interchange`, records information about stations where it is possible to transfer from one line to another. This table has 3 columns: `station`, `line1`, and `line2`. A row in this table indicates that it is possible to transfer from `line1` to `line2` in the station named `station`. Now, consider the following rules that define a `trip`:

```
trip(L,S,E) :- leg(L,S,E).
trip(L,S,E) :- leg(L,S,I), trip(L,I,E).
trip(L,S,E) :- interchange(I,L,M), trip(L,S,I), trip(M,I,E).
```

The first rule states that if there is a direct connection between two stations then a trip can be made between those two stations. The second rule is a recursive rule which allows trips to be made on the same line and the third rule allows trips to be made with a possible change of lines. Write an embedded SQL program which will prompt the user for a starting station and a starting line and print all stations on the train network to which a trip can be made from the starting station and line.