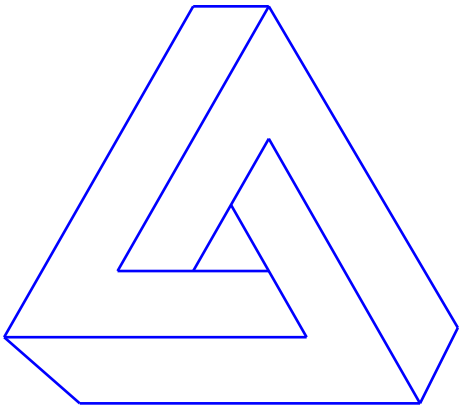
Bases de données

Pro*C

Pascale Brigoulet, Franck Glaziou,
Pascal Lafourcade et Marie-Françoise Servajean



IUT CLERMONT - FD
UNIVERSITÉ
Clermont Auvergne

Nom :

Prénom :

Groupe :

Table des matières

Interface Pro*C

Pro*C est une interface entre le langage C et le SGBD Oracle. Il s'agit d'une interface de précompilation, c'est-à-dire que le programmeur écrit ses ordres SQL dans son code C et ceux-ci sont traduits par un précompilateur. Le rôle du précompilateur est de traduire un programme comprenant des commandes SQL en un programme ne comprenant que des instructions du langage C et pouvant accéder à la base de données (voir Figure 1). Il s'agit de remplacer les commandes SQL par des appels à des modules Oracle combinant ainsi les avantages d'un langage procédural C à ceux d'un langage non procédural SQL.

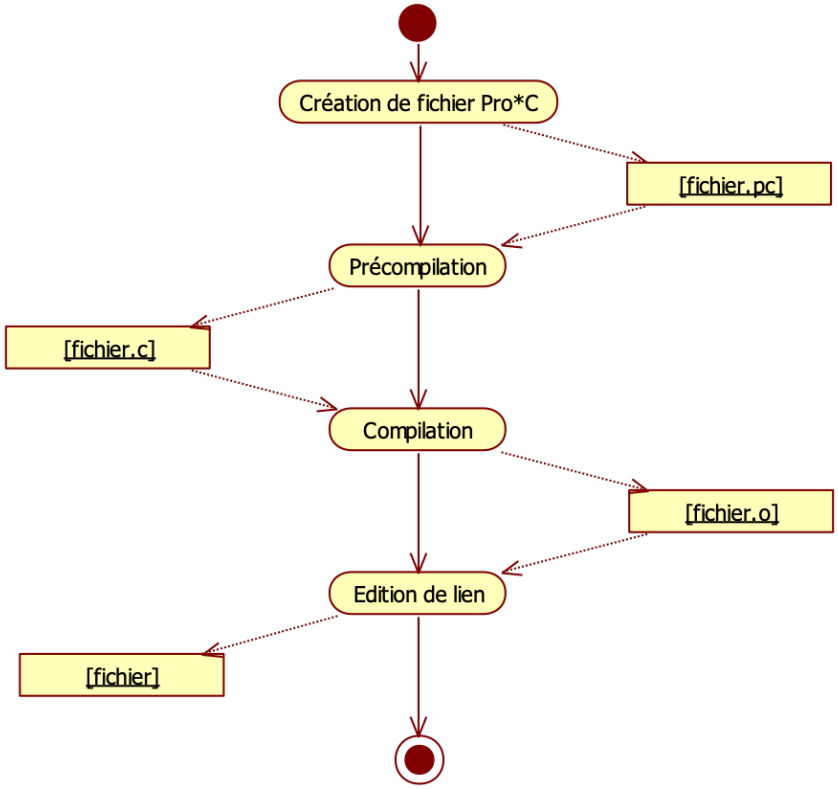


Figure 1 – Illustration du processus permettant d'utiliser un fichier Pro*C

La précompilation, la compilation et l'édition de liens de tels programmes sont masquées par l'exécution d'une commande sur les fichiers sources qui doivent être suffixés par « `.pc` ».

1 Structure d'un programme Pro*C

Un fichier Pro*C se structure de la même manière qu'un fichier C. En plus des commandes SQL, il faut simplement penser à la connexion et la déconnexion à la base de données. Il est donc important de préparer la liaison entre C et Oracle. Pour cela il faut déclarer un ensemble de variables de communication et une zone de communication pour récupérer les comptes rendus Oracle.

1.1 La section `INCLUDE` : (même fonction que le include du C).

Le fichier `SQLCA` permet de connaître le résultat d'une commande Pro*C (erreur, nombre de lignes résultant de la requête, ...). La syntaxe est la suivante : `EXEC SQL INCLUDE SQLCA.H;`

1.2 La déclaration de variables

La déclaration des variables hôtes, c'est-à-dire les variables utilisées dans les commandes SQL, s'effectue exactement de la même manière qu'en C.

```
int pemno;
VARCHAR pname[11];
int pdeptno;
```

Il est important de noter qu'une variable hôte :

— doit être précédée de ':' lorsqu'elle est utilisée (sauf lors de la déclaration),

— ne doit pas être un mot réservé SQL.

Pro*C permet l'utilisation du type `VARCHAR` pour travailler avec les chaînes de caractères de longueur variable. C'est le seul type SQL qui peut être utilisé. Il est important de toujours choisir avec attention les types C car ils vont contenir les informations issues des types SQL. Par exemple, `NUMBER(8,2)` ne peut pas tenir dans un `int`. Il est important de noter que chaque variable de type `VARCHAR` va être traduite par le précompilateur par une structure C. `VARCHAR poste[40];` Le code ci-dessus va être traduit ainsi :

```
struct {
unsigned int len; /* longueur de la chaîne */
unsigned char arr[40]; /* la chaîne elle-même */
} poste ;
```

L'utilisation de `VARCHAR` permet donc de s'abstraire de la gestion du `'\0'` qui n'est pas reconnu par Oracle. Ainsi la longueur de la chaîne est celle située dans l'entier `len`. En sortie d'une commande, Oracle complète la chaîne avec des espaces à droite et met à jour la longueur. Pour entrer une commande, il faut faire attention à cela.

1.3 La connexion à la base de données

Le plus simple est souvent de définir une fonction à réutiliser à chaque fois.

```
void connexion()
{ VARCHAR uid[50];
char login[20];
char passwd[20];
printf("Donner votre login : ");
scanf("%s",login);
printf("\nDonnez votre mot de passe Oracle : ");
scanf("%s",passwd);
printf("\n");
strcpy(uid.arr,login);
strcat(uid.arr,"/");
strcat(uid.arr,passwd);
strcat(uid.arr,"@kirov");
uid.len=strlen(uid.arr);

EXEC SQL CONNECT :uid;
if (sqlca.sqlcode==0)
printf(" Connexion réussie avec succès.\n\n");
```

```
else
{
printf ("Problème à la connexion.\n\n");
exit(1);
}
}
```

Pour la déconnexion, le même principe est à utiliser. Cela permet d'appeler la fonction déclarée à chaque fois que nécessaire. Il est possible de prévoir le cas où il faut valider les transactions et les cas où il faut les annuler.

```
void deconnexion(int validation)
{
if (validation == 1)
{
EXEC SQL COMMIT WORK RELEASE;
} else
{
EXEC SQL ROLLBACK WORK RELEASE;
}
printf("Déconnexion sans problème.\n");
}
```

1.4 Le corps de l'application

Il contient essentiellement des ordres SQL aussi bien pour la manipulation de données que pour la définition des données. La syntaxe utilisée est la même que celle de PL/SQL avec en plus l'utilisation possible de variables hôtes partout où une constante peut être employée (nom de relation, nom d'attribut,...).

```
EXEC SQL CREATE TABLE empTest(empno NUMBER(2));

EXC SQL INSERT INTO tligne VALUES('exception produit inexistant');

EXEC SQL SELECT job INTO :function FROM emp WHERE noemp=301;

EXEC SQL UPDATE emp SET sal = :sal WHERE noemp = :noemp;

EXEC SQL DELETE FROM emp WHERE noemp = :noemp;
```

Toutes les sortes de requêtes peuvent être utilisées. Les requêtes `SELECT` suivent exactement les mêmes règles qu'en PL/SQL. Dans l'exemple précédent, le `SELECT` doit retourner une et une seule ligne. L'utilisation des curseurs pour les multi-lignes est présentée après.

Exercice 1.1.

1. Écrire un programme Pro-C qui se connecte à Kirov puis affiche le nombre de produits contenus dans la base de données, puis se déconnecte.

2. Écrire un programme Pro-C qui demande à l'utilisateur un produit et l'ajoute à la BD.

1.5 Le traitement des erreurs

1.5.1 Le fichier SQLCA.H

Il définit une structure de données 'sqlca' dont les champs sont mis à jour après chaque exécution d'un ordre SQL et qui en donne le compte rendu.

```
struct {
long sqlcode; /* code resultant de l'exécution
=0 -> ok,
>0 -> ok avec un code d'état,
<0 -> erreur */

struct {
unsigned short sqlerrml; /*longueur du message*/
char sqlerrmc[70]; /*message d'erreur*/
} sqlerrm;

long sqlerrd[6]; /* seul sqlerrd[2] est utilisé -> donne le nombre de lignes modifiées
                        UPDATE ou rajoutées par INSERT ou ramenées par un SELECT*/

char sqlwarn[8]; /*sqlwarn[0] 'W' -> warning*/
sqlwarn[0] = ''/*-> pas de warning*/
sqlwarn[1] = 'W'/*-> troncation numérique ou char*/
sqlwarn[2] = 'W'/*-> valeur Null est ignore*/
sqlwarn[3] = 'W'/*-> plus de champs dans SELECT que de variables pour recevoir*/
sqlwarn[4] = 'W'/*-> toutes les lignes d'une table sont touchées (par DELETE ou UPDATE par
               exemple)*/
sqlwarn[5]  /*inutilisé*/
sqlwarn[6] = 'W'/*-> Oracle a dû exécuter un rollback*/
sqlwarn[7] = 'W'/*-> la donnée ramenée par un FETCH a été modifié depuis que la clause
               SELECT a été execute*/
} sqlca;
```

Après chaque commande SQL il est possible de tester le champ de sqlca correspondant à une erreur possible dans le cadre de cette commande.

1.6 La commande WHENEVER

Oracle permet d'utiliser des directives qui spécifient le traitement à effectuer en cas d'erreur.

```
EXEC SQL WHENEVER [SQLERROR|SQLWARNING|NOT FOUND][STOP|CONTINUE|GOTO <label>|DO <fonction>];
```

La valeur du premier paramètre est le type d'erreur à tester :

```
SQLERROR <=> sqlca.sqlcode<0
SQLWARNING <=> sqlca.sqlwarn[0]='W'
NOT FOUND <=> sqlca.sqlcode=+1403 (not row found)
```

La valeur du deuxième paramètre est le type d'action à entreprendre dans le cas où il y a erreur :

— STOP cause l'arrêt du programme avec ROLLBACK ;

— CONTINUE ignore l'erreur et continue le programme ;

— GOTO <label> passe le contrôle du programme au <label> ;

— DO <fonction> appelle une fonction.

La commande WHENEVER doit précéder la commande SQL à tester.

```
EXEC SQL WHENEVER SQLERROR GOTO labx;
EXEC SQL SELECT ...
...
labx : printf ("Programme terminé sur ERREUR ! \n");
printf ("Le sqlcode est %d \n", sqlca.sqlcode);
printf ("Le message SQL est : %70s",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
```

Remarque 1.1. Pour garder une programmation structurée, il est préférable de tester, après chaque commande SQL, la valeur du sqlca.sqlcode plutôt que d'utiliser la commande WHENEVER.

Exercice 1.2. Rajouter aux 2 exercices précédents la gestion des erreurs.

1. Écrire un programme Pro-C qui affiche le nombre de produits contenus dans la BD.

2. Écrire un programme Pro-C qui demande à l'utilisateur un produit et l'ajoute à la BD.

1.6.1 Les indicateurs de variables

À chaque variable hôte il est possible d'associer un indicateur de variable pour connaître ses valeurs particulières. Il est utilisé essentiellement pour travailler avec les valeurs `NULL`. Par exemple lors d'une requête SQL, il indique si le contenu de la variable associée est `NULL` car il n'est pas possible de tester en C si une variable est nulle (cette variable n'est pas un pointeur).

Un indicateur de variable hôte :

— doit être du type `short` ;

— doit être précédé de ':' lorsqu'il est utilisé dans une requête SQL ;

— ne doit pas être un mot réservé SQL ;

— doit être précédé par une variable hôte dans un ordre SQL.

Une variable indicateur a pour valeur :

— $0 \Rightarrow$ de problème (variable `NOT NULL` et non tronquée),

— $-1 \Rightarrow$ la valeur de la variable hôte associée est `NULL`,

— $> 0 \Rightarrow$ la valeur retournée dans la variable hôte est tronquée.

Une variable indicateur peut être utilisée en entrée ou en sortie d'une commande SQL. En sortie elle indique le contenu effectif de la variable associée ; en entrée elle permet de donner la valeur NULL à une variable hôte pour pouvoir, par exemple, insérer des valeurs NULL dans une relation.

```
EXEC SQL SELECT nomemp INTO :nom:indicateur FROM emp;
IF (indicateur == -1) {
//Le nom est NULL dans la base
}
```

Dans un `INSERT`, il faut penser à utiliser le mot clef `INDICATOR : indicateur = -1;`

```
EXEC SQL INSERT INTO emp VALUES(:nom INDICATOR :indicateur);
//Quelque soit le contenu de la variable nom, nous avons inséré NULL
```

1.7 Les SELECT multi-lignes

Dans le cas où la requête retournera un certain nombre de lignes, ou dans le cas où seules les x premières lignes sont souhaitées, il est possible d'utiliser les variables tableaux.

```
VARCHAR nom[100][10];
int empno[100];
float salaire[100];
...
EXEC SQL SELECT nomemp, noemp, sal INTO :nom, :empno, :salaire FROM emp;
```

Cette requête retournera au maximum 100 lignes dans les 100 éléments des tableaux. Le nombre de lignes retournées se trouve dans `sqlca.sqlerrd[2]`. S'il y a plus de 100 lignes et si la même requête est exécutée une deuxième fois alors les mêmes lignes seront récupérées. La clause `FOR` permet d'utiliser un nombre d'éléments différents de la dimension du tableau.

```
EXEC SQL BEGIN DECLARE SECTION;
     int nbMax;
EXEC SQL END DECLARE SECTION;

nbMax = 9;
EXEC SQL FOR :nbMax UPDATE TIMM2018 SET deptimm = '01';
```

Cette requête ne met à jour que les 9 lignes. Dans le cas général où on ne connaît pas le nombre de lignes résultant d'une requête, on doit utiliser un curseur. Comme en PL/SQL, pour utiliser un curseur il y a 4 commandes Pro*C :

— Le curseur est défini par son nom et par la requête qui lui est associée.

```
EXEC SQL DECLARE <nom_curseur> CURSOR FOR SELECT ...FROM ...;
```

— L'ouverture d'un curseur implique la création de la table contenant le résultat de la requête associée à ce curseur. De plus le curseur est positionné sur la première ligne de cette table.

```
EXEC SQL OPEN <nom_curseur>;
```

— L'instruction `FETCH` renvoie dans les variables hôtes les valeurs des attributs de la ligne de la table de travail sur lequel est positionné le curseur puis positionne le curseur sur la prochaine ligne de la table.

```
EXEC SQL FETCH <nom curseur> INTO <liste de variables hôtes>;
```

Si lors d'une instruction `FETCH`, il n'y a plus de lignes dans la table ou si la table est vide, `sqlca.sqlcode` prend la valeur `+1403` qu'il est possible de tester avec l'instruction :

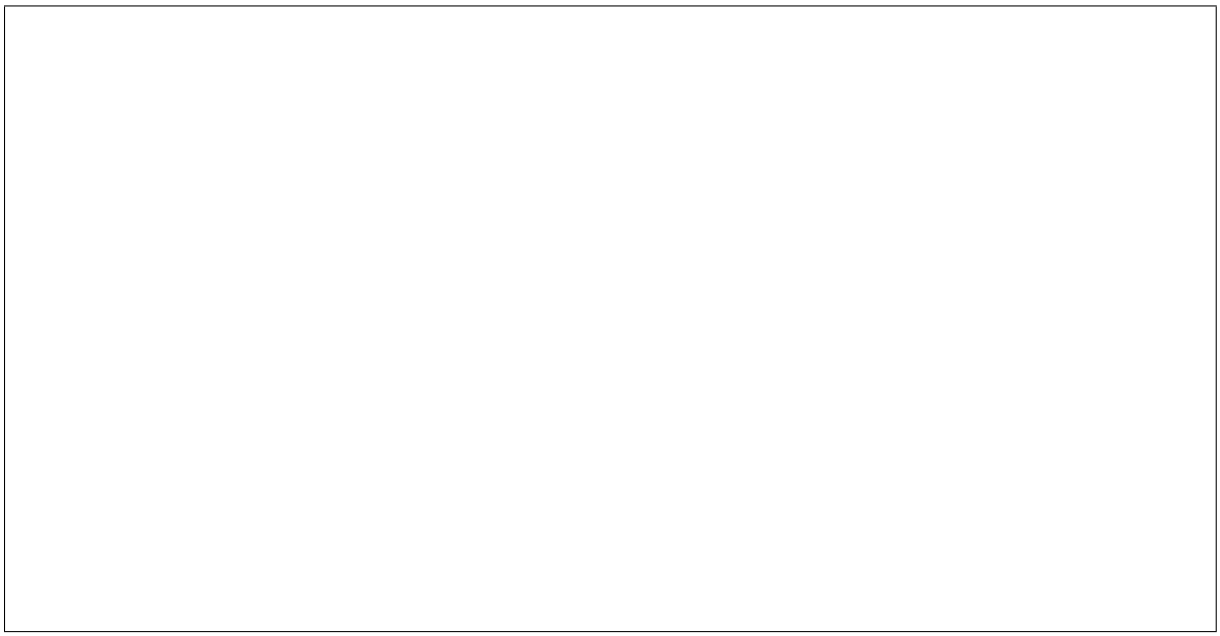
```
EXEC SQL WHENEVER NOT FOUND.
```

— Enfin, l'instruction `CLOSE` ferme le curseur. Toute instruction `FETCH` qui accède à un curseur fermé produit une erreur.

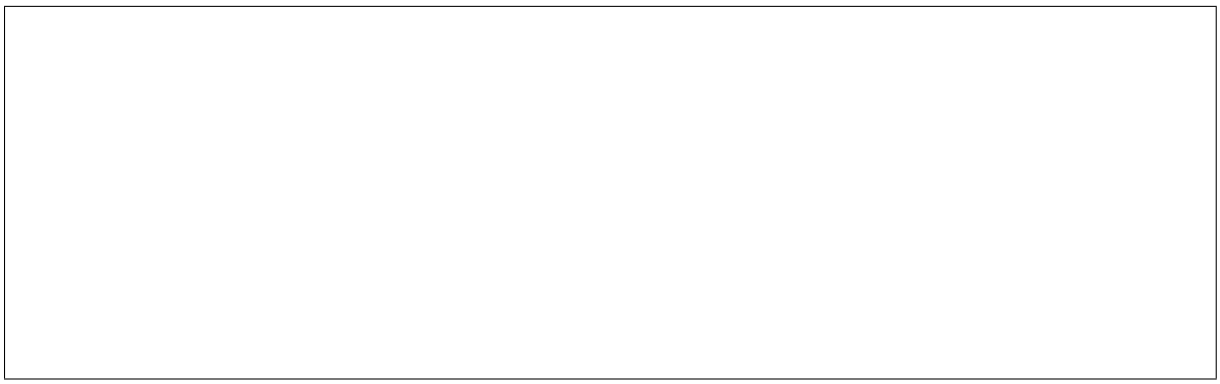
```
EXEC SQL CLOSE <nom du curseur>;
```

Exercice 1.3. 1. Écrire un programme Pro-C qui liste les produits d'un fournisseur saisie par l'utilisateur.

2. Écrire un programme Pro-C qui ajoute un fournisseur.

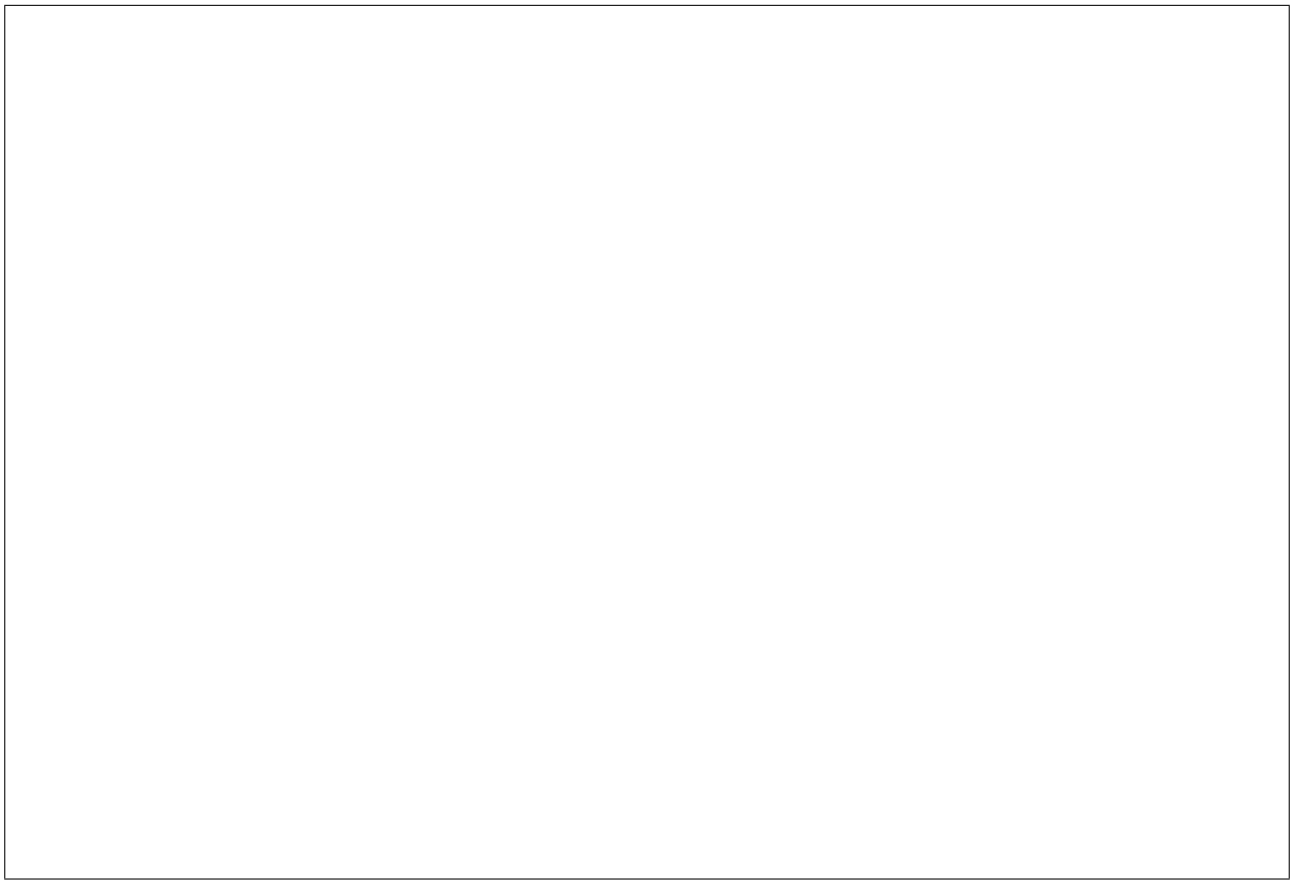


3. Écrire un programme Pro-C qui efface un fournisseur demandé à l'utilisateur et tous les produits d'un fournisseur.



4. Écrire un programme Pro-C qui modifie un produit, en commencant par afficher les anciennes valeurs.

Exercice 1.4 (Indicator)**.** 1. Afficher les numéros de produits et les prix pour un code de rayon saisi par l'utilisateur. Si `prix=NULL` afficher "le prix non saisi".



2. Écrire un programme Pro-C qui ajoute un fournisseur.

3. Afficher par étages la liste des numéro des produits avec leur designation. Si designation est null alors afficher “Pas de designation”.

```
Etage 1
Designation YY n produit XXX
Designation YY n produit XXX

Etage 2
Designation YY n produit XXX
Designation YY n produit XXX
...
```

2 Exemple

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

#define size_t long
EXEC SQL INCLUDE SQLCA.H;
EXEC SQL INCLUDE SQLDA.H;
EXEC SQL INCLUDE ORACA.H;

void connexion()
{ VARCHAR uid[50];
char login[20];
```

```
char passwd[20];
printf("Donner votre login : ");
scanf("%s",login);
printf("\nDonnez votre mot de passe Oracle : ");
scanf("%s",passwd);
printf("\n");
strcpy(uid.arr,login);
strcat(uid.arr,"/");
strcat(uid.arr,passwd);
strcat(uid.arr,"@kirov");
uid.len=strlen(uid.arr);

EXEC SQL CONNECT :uid;
if (sqlca.sqlcode==0)
printf(" Connexion réussie avec succès.\n\n");
else
{
printf ("Problème à la connexion.\n\n");
        exit(1);
        }
}

void deconnexion(int validation)
{
if (validation == 1)
{
EXEC SQL COMMIT WORK RELEASE;
} else {
EXEC SQL ROLLBACK WORK RELEASE;
}
printf("Déconnexion sans problème.\n");
}

void sql_error(char *msg)
{
char err_msg[128];
long buf_len, msg_len;

EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s\n", msg);
buf_len = sizeof (err_msg);
sqlglm(err_msg, &buf_len, &msg_len);

if (msg_len > buf_len)
msg_len = buf_len;

printf("%.*s\n", msg_len, err_msg);
deconnexion(0);
exit(1);
```

```
}

int main(int argc, char** argv)
{
VARCHAR msg_buf[51];

EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error\n");
connexion();
EXEC SQL CREATE TABLE hello_world(msg VARCHAR2(50));
EXEC SQL INSERT INTO hello_world VALUES ('Hello world!');
EXEC SQL COMMIT;
EXEC SQL SELECT msg INTO :msg_buf FROM hello_world WHERE rownum <= 1;

printf("%.*s\n", msg_buf.len,msg_buf.arr);
deconnexion(1);
return(0);
}
```

3 Extrait de la traduction

```
void connexion()
{ /* VARCHAR uid[50]; */
struct { unsigned short len; unsigned char arr[50]; } uid;
char login[20];
char passwd[20];
printf("Donner votre login : ");
scanf("%s",login);
printf("\nDonnez votre mot de passe oracle : ");
scanf("%s",passwd);
printf("\n");
strcpy(uid.arr,login);
strcat(uid.arr,"/");
strcat(uid.arr,passwd);
strcat(uid.arr,"@kirov");
uid.len=strlen(uid.arr);

{  /* EXEC SQL CONNECT :uid; */
 struct sqlexd sqlstm;
 sqlstm.sqlvsn = 12;
 sqlstm.arrsiz = 4;
 sqlstm.sqladtp = &sqladt;
 sqlstm.sqltdsp = &sqltds;
 sqlstm.iters = (unsigned int)10;
 sqlstm.offset = (unsigned int)5;
 sqlstm.cud = sqlcud0;
 sqlstm.sqlest = (unsigned char *)&sqlca;
 sqlstm.sqlety = (unsigned short) 4352;
 sqlstm.occurs = (unsigned int) 0;
 sqlstm.sqhstv[0] = (unsigned char  *)&uid;
```

```
 sqlstm.sqhstl[0] = (unsigned long ) 52;
 sqlstm.sqhsts[0] = (int) 52;
 sqlstm.sqindv[0] = (short *) 0;
 sqlstm.sqinds[0] = (int) 0;
 sqlstm.sqharm[0] = (unsigned long) 0;
 sqlstm.sqadto[0] = (unsigned short) 0;
 sqlstm.sqtdso[0] = (unsigned short) 0;
 sqlstm.sqphsv = sqlstm.sqhstv;
 sqlstm.sqphsl = sqlstm.sqhstl;
 sqlstm.sqphss = sqlstm.sqhsts;
 sqlstm.sqpind = sqlstm.sqindv;
 sqlstm.sqpins = sqlstm.sqinds;
 sqlstm.sqparm = sqlstm.sqharm;
 sqlstm.sqparc = sqlstm.sqharc;
 sqlstm.sqpadto = sqlstm.sqadto;
 sqlstm.sqptdso = sqlstm.sqtdso;
 sqlstm.sqlcmax = (unsigned int) 100;
 sqlstm.sqlcmin = (unsigned int) 2;
 sqlstm.sqlcincr = (unsigned int)1;
 sqlstm.sqlctimeout = (unsigned int) 0;
 sqlstm.sqlcnowait = (unsigned int) 0;
 sqlcxt((void **)0, &sqlctx, &sqlstm, &sqlfpn);
}
if (sqlca.sqlcode==0)
printf(" Connexion réussie avec succès.\n\n");
else
{
printf ("Problème à la connexion.\n\n");
exit(1);
}
}
...
/* EXEC SQL CREATE TABLE hello_world(msg VARCHAR2(50)); */
{
 struct sqlexd sqlstm;
 sqlstm.sqlvsn = 12;
 sqlstm.arrsiz = 4;
 sqlstm.sqladtp = &sqladt;
 sqlstm.sqltdsp = &sqltds;
 sqlstm.stmt = "create TABLE hello_world (msg VARCHAR2(50))";
 sqlstm.iters = (unsigned int) 1;
 sqlstm.offset = (unsigned int) 81;
 sqlstm.cud = sqlcud0;
 sqlstm.sqlest = (unsigned char *)&sqlca;
 sqlstm.sqlety = (unsigned short) 4352;
 sqlstm.occurs = (unsigned int) 0;
 sqlcxt((void **)0, &sqlctx, &sqlstm, &sqlfpn);
 if (sqlca.sqlcode < 0) sql_error("ORACLE error\n");
}
```

Commandes utiles pour les TP

Première séance : Connexion : `sqlplus <user>/<password>@KIROV`
`source /etc/profile`
Lancer `sqlplus` avec cette commande `rlwrap sqlplus` vous permet d'avoir l'historique.
Lors de la première connexion modifier le mot de passe avec la commande SQL : `PASSWORD;`
Ce qui est équivalent à : `ALTER USER dupond IDENTIFIED BY password;`
Si le mot de passe est égaré, il faut se connecter en `ssh` sur `londres` et faire `oracle_passwd`
Pour lancer un fichier `.sql` en sqlplus, il suffit de taper : `@toto.sql;`
Pour quitter sqlplus, il suffit de taper : `quit;`

Mise en forme : Sous SQL/PLUS :

```
Set linesize 150 -- positionne la taille d'une ligne
Set pagesize 300 --  positionne le nombre de lignes avant de réafficher les entêtes
Set pages 0 --  n'affiche pas les entêtes
Col <nom_colonne> for A10 --  défini que la colonne nom_colonne va être affiché sur 10 ca$
alphanumériques, 999.99 pour les valeurs numériques.
```

Corbeille : Vider les tables `BIN$$xxxx` : avec la commande `PURGE RECYCLEBIN;`

Débug : Afficher la strucutre de la table `nba` : `describle nba;`
Connaître l'utilisateur connecté : `show user;`
Liste des tables d'un `user` : `SELECT table_name FROM user_tables;`
Lister les tables accessibles par l'utilisateur : `SELECT table_name FROM all_tables;`
Lister toutes les tables possédées par un utilisateur `SELECT * FROM all_tables WHERE owner='PALAFOUR';`
Liste des vues d'un `user` : `SELECT view_name FROM user_views;`
Liste des contraintes : `SELECT * FROM user_constraints WHERE table_name='<table>';`
`SELECT * FROM user_cons_columns WHERE table_name='<table>";`
`show errros` affiche les erreurs des fonctions.

Contraintes : Description d'une table : `describe <table>` ou `desc <table>` ;
Liste des colonnes concernées par les contraintes : `SELECT * FROM user_cons_columns;`
Lire une table d'un autre schéma :
`SELECT * FROM <schema>.<table_name>; -- ou schma = login de connexion de l'utilisateur`

Affichage : `SET HEADING OFF`
`SET FEEDBACK OFF`

Travailler à la maison Il est possible d'accèder par SSH à la machine `londres`. L'adresse de la passerelle est `ssh.iut-clermont.uca.fr` qui n'est accessible que par une authentification avec des clefs SSH.

Cet accès vous permettra d'accèder au serveur de base de données comme lorsque vous travailler à l'IUT. Avant de pouvoir se connecter, il faut activer l'accès à l'IUT sur `http://berlin.iut.local` ce qui peut prendre 30 minutes.

Pour déposer vos clefs vous pouvez utiliser `https://homeweb.iut-clermont.uca.fr`

En cas de blocage : Dans un terminal executer la commande suivante pour voir les `pid` des processus zombies : `ps -aux | grep sqlplus`
Ensuite tuer ces zombies grâce à la commande `kill -9 numerodepid`