Interfaçage de programmation

© Olivier Caron

Polytech'Lille

Le SGBD est-il suffisant ? (1/2)

√ Les pour :

- La puissance du langage de requêtes.

Le SGBD est-il suffisant ? (1/2)

√ Les pour :

- La puissance du langage de requêtes.
- L'aspect simplification (via les vues)

Le SGBD est-il suffisant ? (1/2)

√ Les pour :

- La puissance du langage de requêtes.
- L'aspect simplification (via les vues)
- Les outils 4GL (SQL Forms Oracle, pgaccess (en cours), Ms Access,. . .)
- Les procédures stockées (cf deuxième année) : premier lien entre programmation et bases de données.

Le SGBD est-il suffisant ? (2/2)

√ Les contre :

- Une action se décompose en un séquencement d'étapes
  exemple : emprunt d'un livre

Le SGBD est-il suffisant ? (2/2)

√ Les contre :

- Une action se décompose en un séquencement d'étapes
  exemple : emprunt d'un livre
- Problème de la cloture transitive

Le SGBD est-il suffisant ? (2/2)

√ Les contre :

- Une action se décompose en un séquencement d'étapes
  exemple : emprunt d'un livre
- Problème de la cloture transitive
- Interface Homme-Machine

Le SGBD est-il suffisant ? (2/2)

√ Les contre :

- Une action se décompose en un séquencement d'étapes
  exemple : emprunt d'un livre
- Problème de la cloture transitive
- Interface Homme-Machine
- Accès multi-bases

Point de vue des applications

√ Objectif : Ecrire une application quelconque

Point de vue des applications

- √ Objectif : Ecrire une application quelconque
- √ Une contrainte : certaines données doivent être persistantes.
- √ Plus facile d'utiliser une API vers un SGBD que de gérer la persistance via des fichiers.

Interface Langage et Bases de données

√ Le Top ! : API normalisée

- Un même programme pour plusieurs SGBDs
- Ex : JDBC, ODBC, ANSI embedded C

Interface Langage et Bases de données

- √ Le Top ! : API normalisée
  - ▶ Un même programme pour plusieurs SGBDs
  - ▶ Ex : JDBC, ODBC, ANSI embedded C
- √ Le compromis :
  - ▶ Langage de programmation standard (C, Java, Pascal,. . .)

Interface Langage et Bases de données

- √ Le Top ! : API normalisée
  - Un même programme pour plusieurs SGBDs
  - Ex : JDBC, ODBC, ANSI embedded C
- √ Le compromis :
  - Langage de programmation standard (C, Java, Pascal,. . .)
  - API accès bases de données

Interface Langage et Bases de données

- √ Le Top ! : API normalisée
  - Un même programme pour plusieurs SGBDs
  - Ex : JDBC, ODBC, ANSI embedded C
- √ Le compromis :
  - Langage de programmation standard (C, Java, Pascal,. . .)
  - API accès bases de données
  - Possibilité d'exploiter le langage SQL

Interface Langage et Bases de données

- √ Le Top ! : API normalisée
  - ▶ Un même programme pour plusieurs SGBDs
  - ▶ Ex : JDBC, ODBC, ANSI embedded C
- √ Le compromis :
  - ▶ Langage de programmation standard (C, Java, Pascal,. . .)
  - ▶ API accès bases de données
  - ▶ Possibilité d'exploiter le langage SQL
- √ Les mauvais : le tout propriétaire

Le cas Postgres

√ Reconnait JDBC, ODBC, embedded C

Le cas Postgres

- √ Reconnait JDBC, ODBC, embedded C
- √ ODBC est considéré plus comme un langage intermédiaire entre applications , moins comme une interface de programmation.

Le cas Postgres

- √ Reconnait JDBC, ODBC, embedded C
- √ ODBC est considéré plus comme un langage intermédiaire entre applications , moins comme une interface de programmation.
- √ Compatible PHP

Le cas Postgres

- √ Reconnait JDBC, ODBC, embedded C
- √ ODBC est considéré plus comme un langage intermédiaire entre applications , moins comme une interface de programmation.
- √ Compatible PHP
- √ Procédures Stockées (PLPGSQL)

Le cas Postgres

- √ Reconnait JDBC, ODBC, embedded C
- √ ODBC est considéré plus comme un langage intermédiaire entre applications , moins comme une interface de programmation.
- √ Compatible PHP
- √ Procédures Stockées (PLPGSQL)
- √ Interfaces possibles C (lib postgres, pgeasy, embedded C), C++, Python, TCL-TK, Perl.

Une application bases de données (1/2)

√ Structuration du programme :

1. Connexion à une base : nom du serveur, nom de la base, nom de l'utilisateur, mot de passe

Une application bases de données (1/2)

√ Structuration du programme :

1. Connexion à une base : nom du serveur, nom de la base, nom de l'utilisateur, mot de passe
2. Etablissement de requête (SQL)

Une application bases de données (1/2)

√ Structuration du programme :

1. Connexion à une base : nom du serveur, nom de la base, nom de l'utilisateur, mot de passe
2. Etablissement de requête (SQL)
3. Traitement de la réponse

Une application bases de données (1/2)

√ Structuration du programme :

1. Connexion à une base : nom du serveur, nom de la base, nom de l'utilisateur, mot de passe
2. Etablissement de requête (SQL)
3. Traitement de la réponse
4. Si non fini, retour au point 2

Une application bases de données (1/2)

√ Structuration du programme :

1. Connexion à une base : nom du serveur, nom de la base, nom de l'utilisateur, mot de passe
2. Etablissement de requête (SQL)
3. Traitement de la réponse
4. Si non fini, retour au point 2
5. Déconnexion

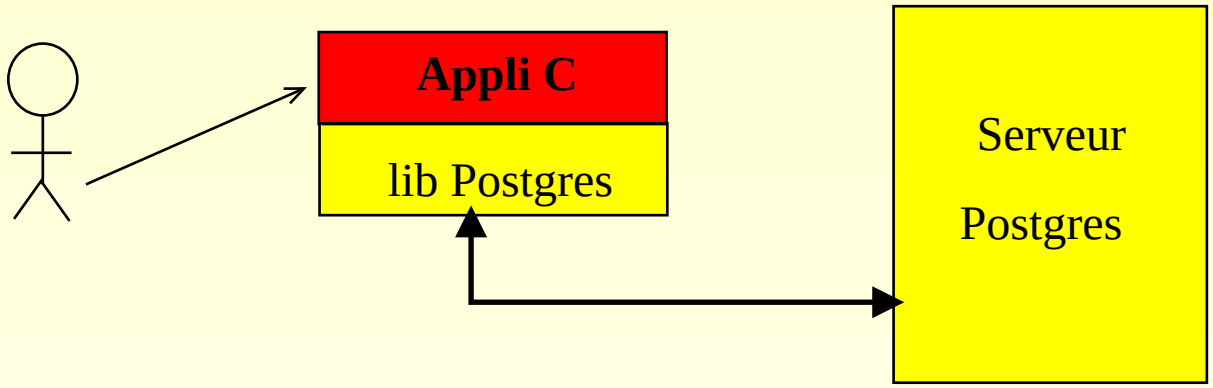
Une application bases de données (2/2)

- √ Aspects réseau, sécurité,. . . : toujours vérifier la validité de la commande envoyée

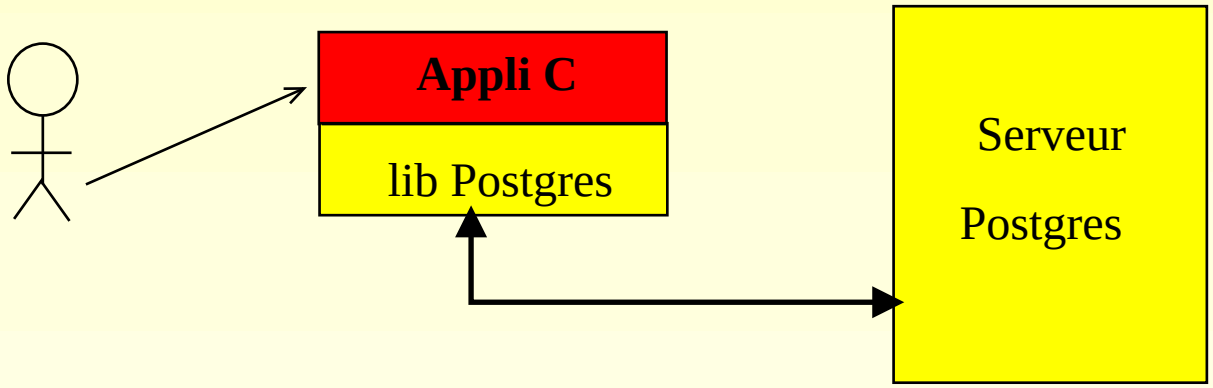
Une application bases de données (2/2)

- √ Aspects réseau, sécurité,. . . : toujours vérifier la validité de la commande envoyée
- √ Variante possible de la structuration d'un programme : accès multi-bases.
  - ▶ Les requêtes doivent être paramétrées par la connexion.

La librairie Postgres Query



La librairie Postgres Query



√ Réseau Polytech'Lille :

- `/usr/include/postgresql/libpq-fe.h`
- `/usr/lib/libpq.a` ou `/usr/lib/libpq.so`

Lib. Postgres : variables d'environnement

√ Les variables d'environnement suivantes peuvent être utilisées afin de faciliter l'écriture des programmes.

nom	**désignation**
PGHOST	nom du serveur
PGOPTIONS	options pour le serveur distant postgres
PGPORT	numéro du port TCP utilisé
PGTTY	fichier ou device tty pour l'affichage des erreurs
PGDATABASE	nom de la base de données

Lib. Postgres : établissement d'une connexion (1/2)

```
PGconn *PQsetdbLogin(char *pghost, char *pgport,
                     char *pgoptions, char *pgtty,
                     char *dbname,char *login, char *password) ;
```

√ Si un argument est positionné à `NULL` alors la variable d'environnement correspondante est recherchée.

Lib. Postgres : établissement d'une connexion (1/2)

```
PGconn *PQsetdbLogin(char *pghost, char *pgport,
                     char *pgoptions, char *pgtty,
                     char *dbname,char *login, char *password) ;
```

- √ Si un argument est positionné à `NULL` alors la variable d'environnement correspondante est recherchée.
- √ Si cette variable n'est pas positionnée, les valeurs par défaut du système sont utilisées.

Lib. Postgres : établissement d'une connexion (2/2)

√ `PQsetdbLogin` retourne un pointeur sur une structure de nom `PGconn`.

Lib. Postgres : établissement d'une connexion (2/2)

- √ `PQsetdbLogin` retourne un pointeur sur une structure de nom `PGconn`.
- √ La fonction `PQstatus` permet de s'assurer de la réalisation de la connexion : deux valeurs de retour sont possibles : `CONNECTION_OK` et `CONNECTION_BAD`.

Lib. Postgres : établissement d'une connexion (2/2)

- √ `PQsetdbLogin` retourne un pointeur sur une structure de nom `PGconn`.
- √ La fonction `PQstatus` permet de s'assurer de la réalisation de la connexion : deux valeurs de retour sont possibles : `CONNECTION_OK` et `CONNECTION_BAD`.
- √ La fonction `PQerrorMessage` affiche le message d'erreur associé à la connexion.

Lib. Postgres : établissement d'une connexion (2/2)

- √ `PQsetdbLogin` retourne un pointeur sur une structure de nom `PGconn`.
- √ La fonction `PQstatus` permet de s'assurer de la réalisation de la connexion : deux valeurs de retour sont possibles : `CONNECTION_OK` et `CONNECTION_BAD`.
- √ La fonction `PQerrorMessage` affiche le message d'erreur associé à la connexion.
- √ Par la suite, la variable de type `PGconn` sera utilisée pour chaque accès à la base de données distante.

Lib. Postgres : Validité de la connexion

Signature	Désignation
`char *PQdb(PGconn *conn);`	nom de la base
`char *PQhost(PGconn *conn);`	nom du serveur

`char *PQoptions(PGconn *conn);`
`char *PQport(PGconn *conn);`
`char *PQtty(PGconn *conn) ;`
	les options
	port tcp
	nom du terminal

`ConnStatusType *PQstatus(PGconn *conn);`	état
`char *PQerrorMessage(PGconn *conn);`	message d'erreur

Lib. Postgres : exécution de commandes SQL

- √ Une nouvelle structure C est utilisée : `PGresult`.
- √ Cette structure contient toutes les informations nécessaires à l'exploitation des résultats d'une requête SQL réalisée par l'intermédiaire de la routine `PQexec`.

```
PGresult *PQexec(PGconn *conn,char *cmdeSQL) ;
```

Lib. Postgres : exécution de commandes SQL

- √ Une nouvelle structure C est utilisée : `PGresult`.
- √ Cette structure contient toutes les informations nécessaires à l'exploitation des résultats d'une requête SQL réalisée par l'intermédiaire de la routine `PQexec`.

```
PGresult *PQexec(PGconn *conn,char *cmdeSQL) ;
```

- √ Exploitez la fonction `sprintf` !

Lib. Postgres : validité de la requête

√ Avant d'exploiter le résultat d'une requête, il faut vérifier le bon déroulement de commande SQL.

Lib. Postgres : validité de la requête

- √ Avant d'exploiter le résultat d'une requête, il faut vérifier le bon déroulement de commande SQL.
- √ Une fonction d'accusé de réception :

```
ExecStatusType PQresultStatus(PGresult *res) ;
```

Lib. Postgres : validité de la requête

- √ Avant d'exploiter le résultat d'une requête, il faut vérifier le bon déroulement de commande SQL.
- √ Une fonction d'accusé de réception :
  `ExecStatusType PQresultStatus(PGresult *res) ;`
- √ Le type énuméré `ExecStatusType` peut prendre les valeurs :
  `PGRES_COMMAND_OK` (la requête est une commande),
  `PGRES_TUPLES_OK`, `PGRES_EMPTY_QUERY`, `PGRES_BAD_RESPONSE`,
  `PGRES_NONFATAL_ERROR`, `PGRES_FATAL_ERROR`.
- √ La fonction `char *PQcmdStatus(PGresult *res)` permet d'afficher l'état de la dernière commande.

Lib. Postgres : traitement résultat Requête

√ Analyse de la structure `PGresult` :

Signature et Désignation
`int PQntuples(PGresult *res)` → nbre de tuples
`int PQnfields(PGresult *res)` → nbre d'attributs
`char *PQfname(PGresult *res,int field_index)`
→nom des attributs
`char *PQfnumber(PGresult *res, char *field_name)`
→ index de l'attribut
`char *PQgetvalue(PGresult *res,int tup_num,int field_num)`
→ valeur de l'attribut

√ L'index commence à partir de la valeur 0

Lib. Postgres : traitement de plusieurs requêtes

√ Il est nécessaire de libérer la mémoire associée à une variable de type `PGresult` quand on ne l'utilise plus ou quand **on l'utilise pour une autre requête**.

Lib. Postgres : traitement de plusieurs requêtes

- √ Il est nécessaire de libérer la mémoire associée à une variable de type `PGresult` quand on ne l'utilise plus ou quand **on l'utilise pour une autre requête**.
- √ Cela est réalisé par la procédure :
  `void PQclear(PQresult *res)`

Lib. Postgres : fermeture de la connexion

√ Les deux fonctions suivantes permettent de terminer une connexion à une base de données distante :

- `void PQfinish(PGconn *conn);` : termine la connexion.
- `void PQreset(PGconn *conn);` : réinitialise une connexion.

Lib. Postgres pgeasy

√ Couche logicielle supplémentaire

Lib. Postgres pgeasy

- √ Couche logicielle supplémentaire
- √ Basée sur la lib. Postgres

Lib. Postgres pgeasy

- √ Couche logicielle supplémentaire
- √ Basée sur la lib. Postgres
- √ Fonctions plus simples : (moins de paramètres)

Lib. Postgres pgeasy

- √ Couche logicielle supplémentaire
- √ Basée sur la lib. Postgres
- √ Fonctions plus simples : (moins de paramètres)
- √ Couvre les requêtes élémentaires

pgeasy : Un exemple (1/2)

```
#include <stdio.h>
#include <libpq-fe.h>
#include <libpgeasy.h>

int main(void) {
   char cmdeSQL[256], ligne[1000] ;

   connectdb("dbname=test");
   printf("SQL ?") ; scanf("%s", cmdeSQL) ;
```

pgeasy : Un exemple (2/2)

```
    doquery(cmdeSQL) ;
    while (fetch(ligne) != END_OF_TUPLES)
      printf("%s\n", ligne);
    disconnectdb();
    return 0;
}
```

pgeasy : en un coup d'œil

```
PGresult   *doquery(char *query);
PGconn     *connectdb(char *options);
void  disconnectdb(void);
int        fetch(void *param,...);
int        fetchwithnulls(void *param,...);
void       on_error_continue(void);
void       on_error_stop(void);
PGresult   *get_result(void);
void       set_result(PGresult *newres);
void       unset_result(PGresult *oldres);
void       reset_fetch(void);

#define END_OF_TUPLES   (-1)
```

pgeasy : commentaires

√ API sommaire !

pgeasy : commentaires

- √ API sommaire !
- √ Les inconvénients d'un langage de script (absence de typage). . .sans les avantages (compilation nécessaire)

pgeasy : commentaires

- √ API sommaire !
- √ Les inconvénients d'un langage de script (absence de typage). . .sans les avantages (compilation nécessaire)
- √ nécessite lib. Postgres !

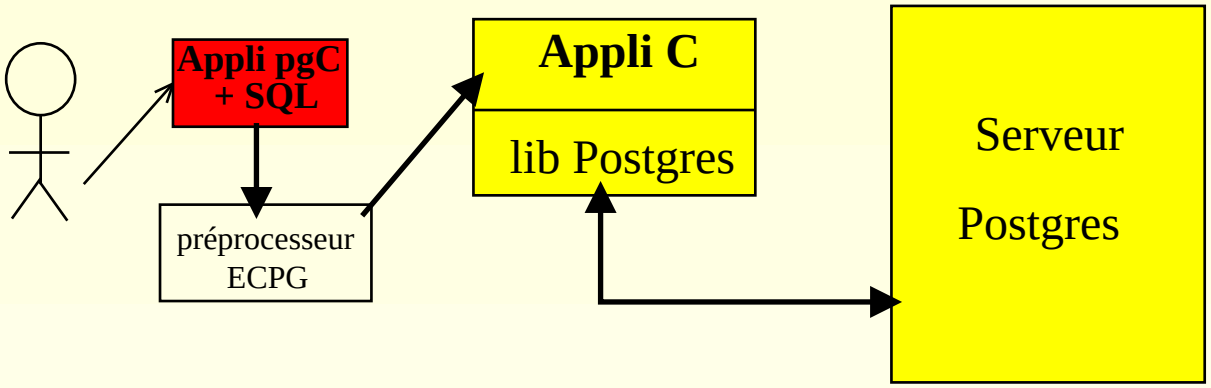
Embedded C (ECPG)

$\surd$ Normalisation ANSI $\rightarrow$ applications non liés à un SGBD

Embedded C (ECPG)

√ Normalisation ANSI → applications non liés à un SGBD

√ Principe :



ECPG : compilation

√ Préprocesseur : ecpg

ECPG : compilation

√ Préprocesseur : ecpg

√ Même principe que le préprocesseur C

ECPG : compilation

- √ Préprocesseur : ecpg
- √ Même principe que le préprocesseur C
- √ Directive `EXEC SQL`

ECPG : compilation

- √ Préprocesseur : ecpg
- √ Même principe que le préprocesseur C
- √ Directive `EXEC SQL`
- √ Librairies : `libecpg.a` ou `libecpg.so`

ECPG : sections de déclaration

√ Sections délimitées par :

```
exec sql begin declare section ;
  ...
exec sql end declare section ;
```

ECPG : sections de déclaration

- Sections délimitées par :

```
exec sql begin declare section ;
  ...
exec sql end declare section ;
```

- Uniquement déclaration de variables

ECPG : sections de déclaration

- √ Sections délimitées par :

```
exec sql begin declare section ;
  ...
exec sql end declare section ;
```

- √ Uniquement déclaration de variables
- √ Variables partagées entre le préprocesseur et le code C.
- √ Lien entre types SQL et types C.

ECPG : section d'inclusion

√ Notation :`exec sql include` **nomFichier** ;

ECPG : section d'inclusion

- √ Notation :`exec sql include` **nomFichier** ;
- √ Ca ne génère pas : `#include` **<nomFichier>**
- √ Cette section copie le contenu du fichier dans le code C résultat

ECPG : section d'inclusion

- √ Notation :`exec sql include` **nomFichier** ;
- √ Ca ne génère pas : `#include` **<nomFichier>**
- √ Cette section copie le contenu du fichier dans le code C résultat
- √ Peut donc être utilisé pour spécifier des commandes `exec sql`

ECPG : section d'inclusion

- √ Notation :`exec sql include` **nomFichier** ;
- √ Ca ne génère pas : `#include` **<nomFichier>**
- √ Cette section copie le contenu du fichier dans le code C résultat
- √ Peut donc être utilisé pour spécifier des commandes `exec sql`
- √ Pour détecter les erreurs, insérer dans le code C, la ligne :

```
exec sql include sqlca;
```

ECPG : section de connexion

√ Notation : `exec sql connect to connection target ;`

ECPG : section de connexion

- √ Notation : `exec sql connect to connection target ;`
- √ Etablit une connexion à la base distante

ECPG : section de connexion

- √ Notation : `exec sql connect to connection target ;`
- √ Etablit une connexion à la base distante
- √ `connection target` peut prendre les syntaxes suivantes :

```
dbname[@server][:port][as connection name][user ...]
tcp:postgresql://server[:port][/dbname]
                [as connection name][user ...]
unix:postgresql://server[:port][/dbname]
                 [as connection name][user ...]
user peut prendre les syntaxes suivantes:
user userid
user userid/password
user userid using password
```

ECPG : section de déconnexion

√ Notation : `exec sql disconnect` [**connection target**] ;

ECPG : section de déconnexion

- √ Notation : `exec sql disconnect` [**connection target**] ;
- √ **connection target** peut prendre les formes : connection name, current , all.

ECPG : Exécution de requêtes

√ Notation : `exec sql` *`requete SQL`*

ECPG : Exécution de requêtes

- √ Notation : `exec sql` *`requete SQL`*
- √ Notation : `exec sql at nom_connection` *`requete SQL`*

ECPG : Exécution de requêtes

- √ Notation : `exec sql` *`requete SQL`*
- √ Notation : `exec sql at nom_connection` *`requete SQL`*
- √ Tous les types de requêtes possibles (create, insert, . . .)

ECPG : Exécution de requêtes

- √ Notation : `exec sql` *`requete SQL`*
- √ Notation : `exec sql at nom_connection` *`requete SQL`*
- √ Tous les types de requêtes possibles (create, insert, . . .)
- √ Structure C `sqlca` donne des informations sur l'exécution de la requête
  Exemple : `sqlca.sqlerrd[1]` indique le nombre de lignes créées, modifiées,. . .

ECPG : Exécution de requêtes

- √ Notation : `exec sql` *`requete SQL`*
- √ Notation : `exec sql at nom_connection` *`requete SQL`*
- √ Tous les types de requêtes possibles (create, insert, . . .)
- √ Structure C `sqlca` donne des informations sur l'exécution de la requête
  Exemple : `sqlca.sqlerrd[1]` indique le nombre de lignes créées, modifiées,. . .
- √ Possibilité de préparer, paramétrer des requêtes via des variables

ECPG : récupération des données une ligne résultat

√ Utilisation : `select ...into :nom_variable1, :nom_variable2, ...`

ECPG : récupération des données une ligne résultat

- √ Utilisation : `select ...into :nom_variable1, :nom_variable2, ...`
- √ Seules les variables C définies dans les sections de déclaration sont utilisables.

ECPG : récupération des données une ligne résultat

- √ Utilisation : `select ...into :nom_variable1, :nom_variable2, ...`
- √ Seules les variables C définies dans les sections de déclaration sont utilisables.
- √ Ne fonctionne que lorsqu'il y a qu'une ligne résultat

ECPG : récupération de plusieurs lignes de données

√ Utilisation de curseurs, principe :

1. Déclaration d'un curseur pour une requête

```
exec sql declare nomCurseur cursor for
select ...
```

ECPG : récupération de plusieurs lignes de données

√ Utilisation de curseurs, principe :

1. Déclaration d'un curseur pour une requête
   ```
   exec sql declare nomCurseur cursor for
   select ...
   ```
2. Ouverture du curseur : `exec sql open nomCurseur`

ECPG : récupération de plusieurs lignes de données

√ Utilisation de curseurs, principe :

1. Déclaration d'un curseur pour une requête
   ```
   exec sql declare nomCurseur cursor for
   select ...
   ```
2. Ouverture du curseur : `exec sql open nomCurseur`
3. Disposer d'un mécanisme pour quitter la future boucle de lecture :
   `exec sql whenever not found do break ;`

ECPG : récupération de plusieurs lignes de données

√ Utilisation de curseurs, principe :

1. Déclaration d'un curseur pour une requête
   ```
   exec sql declare nomCurseur cursor for
   select ...
   ```
2. Ouverture du curseur : `exec sql open nomCurseur`
3. Disposer d'un mécanisme pour quitter la future boucle de lecture :
   `exec sql whenever not found do break ;`
4. Dans boucle de lecture, lecture d'une ligne :
   `exec sql fetch from nomCurseur into :nom_var1,...`

ECPG : récupération de plusieurs lignes de données

√ Utilisation de curseurs, principe :

1. Déclaration d'un curseur pour une requête
   ```
   exec sql declare nomCurseur cursor for
   select ...
   ```
2. Ouverture du curseur : `exec sql open nomCurseur`
3. Disposer d'un mécanisme pour quitter la future boucle de lecture :
   `exec sql whenever not found do break ;`
4. Dans boucle de lecture, lecture d'une ligne :
   `exec sql fetch from nomCurseur into :nom_var1,...`
5. Fermeture du curseur :`exec sql close nomCurseur`

ECPG : un exemple complet (1/5)

√ Ouverture de la base `template1` sur weppes, calcul du nombre d'utilisateurs et affichage des noms de ces utilisateurs.

ECPG : un exemple complet (1/5)

- √ Ouverture de la base `template1` sur weppes, calcul du nombre d'utilisateurs et affichage des noms de ces utilisateurs.
- √ le nom et le mot de passe de l'utilisateur courant est passé en paramètre.

ECPG : un exemple complet (2/5)

```
/*************************************************************/
/* exempleECPG.pgc                                           */
/* lit sur la ligne de commande :                            */
/* le nom d'un utilisateur et son mot de passe               */
/* affiche ensuite le nombre de lignes de pg_user            */
/* ainsi que le champ usename de cette table                 */
/*************************************************************/

int main(int argc, char * argv[]) {

  exec sql include sqlca ;  /* gestion des codes d'erreurs */
```

ECPG : un exemple complet (3/5)

```
exec sql begin declare section ;
  char user[30]
  char password[30]  ;
  int nbre ;
  varchar usename[8] ;
exec sql end declare section ;

if (argc != 3) {
  fprintf(stderr,"erreur, usage: %s user passwd\n", argv[0]) ;
  return 1 ;
}
strcpy(user, argv[1]) ;
strcpy(password,argv[2]) ;
```

ECPG : un exemple complet (4/5)

```
exec sql connect to template1@weppes :5432 as connect_weppes
  user :user using :password ;

exec sql at connect_weppes
  select count(*) into :nbre from pg_user ;

printf("Nombre d'utilisateurs : %d\n",nbre) ;
```

ECPG : un exemple complet (5/5)

```
  exec sql declare monCurseur cursor for
    select usename from pg_user ;

  exec sql open monCurseur ;
  exec sql whenever not found do break ;

  while (1) {
    exec sqlfetch from  monCurseur into :usename ;
    printf(" nom user : %s\n", usename) ;
  }

  exec sql close monCurseur ;
  exec sql disconnect connect_weppes ;
}
```