Crossing makes the Future.

PostgreSQL Conference Japan 2024

今、改めてECPGを使ってみる

ＮＴＴテクノクロス株式会社

原田　登志

NTTテクノクロス

自己紹介

- 名前：原田　登志
- 別名：ぬこ@横浜（@nuko_yokohama）
- 所属：ＮＴＴテクノクロス株式会社
- 仕事：PostgreSQLに関するあれこれ
- 「ラーメン PostgreSQL」で検索してください。

【宣伝】弊社サービスの紹介

- NTTテクノクロス社のPostgreSQL関連サービスの紹介

- 移行サービス
  - https://www.ntt-tx.co.jp/products/osscloud/service/migration/
  - OracleDBからPostgreSQL（Aurora等含む）へのDB移行
  - 事例1：移行フェーズ　DB移行実現性検証(PoC)
  - 事例2：開発、移行フェーズ　EC2からAurora PostgreSQLへの移行
- 定額制チケットサービス
  - https://www.ntt-tx.co.jp/products/osscloud/service/ticket/
  - OSS製品検討の支援
  - 保守サービス・トラブルシュート

この発表の目的

- ECPGとはどういうものか理解してもらう
- ECPGプログラムの基本的な組み方を理解してもらう
- OracleのPro*Cプログラムからの移行にECPGが使えることを理解してもらう

埋め込みSQL/ECPG

埋め込みSQLとは

主要DBMSのサポート状況

ECPG

ECPGによるプリプロセス

ECPGヒストリー

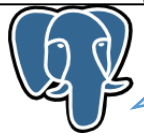
埋め込みSQLとは

- 埋め込みSQL（Embedded SQL）
  - 手続き型プログラミングに、RDBMSを操作するためのSQLを組み込む手法。
  - プログラマはソースコード内部に埋め込みSQLステートメントを直接記述することができるようになる。
- 標準SQLで規定されている。

埋め込みSQLとは

- 埋め込みSQL（C言語）の例。

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

EXEC SQL BEGIN DECLARE SECTION;
char* connection_string = "dbname=mydb user=myuser";
EXEC SQL END DECLARE SECTION;

int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL BEGIN DECLARE SECTION;
    int employee_id = 1;
    char employee_name[50];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT name INTO :employee_name FROM employees WHERE id = :employee_id;

    if (sqlca.sqlcode < 0) {
        printf("Failed to fetch employee name¥n");
        exit(1);
    }

    printf("Employee name: %s¥n", employee_name);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;

    return 0;
}
```



Cソース内に
SQLを
埋め込んでいる

主要DBMSでのサポート状況

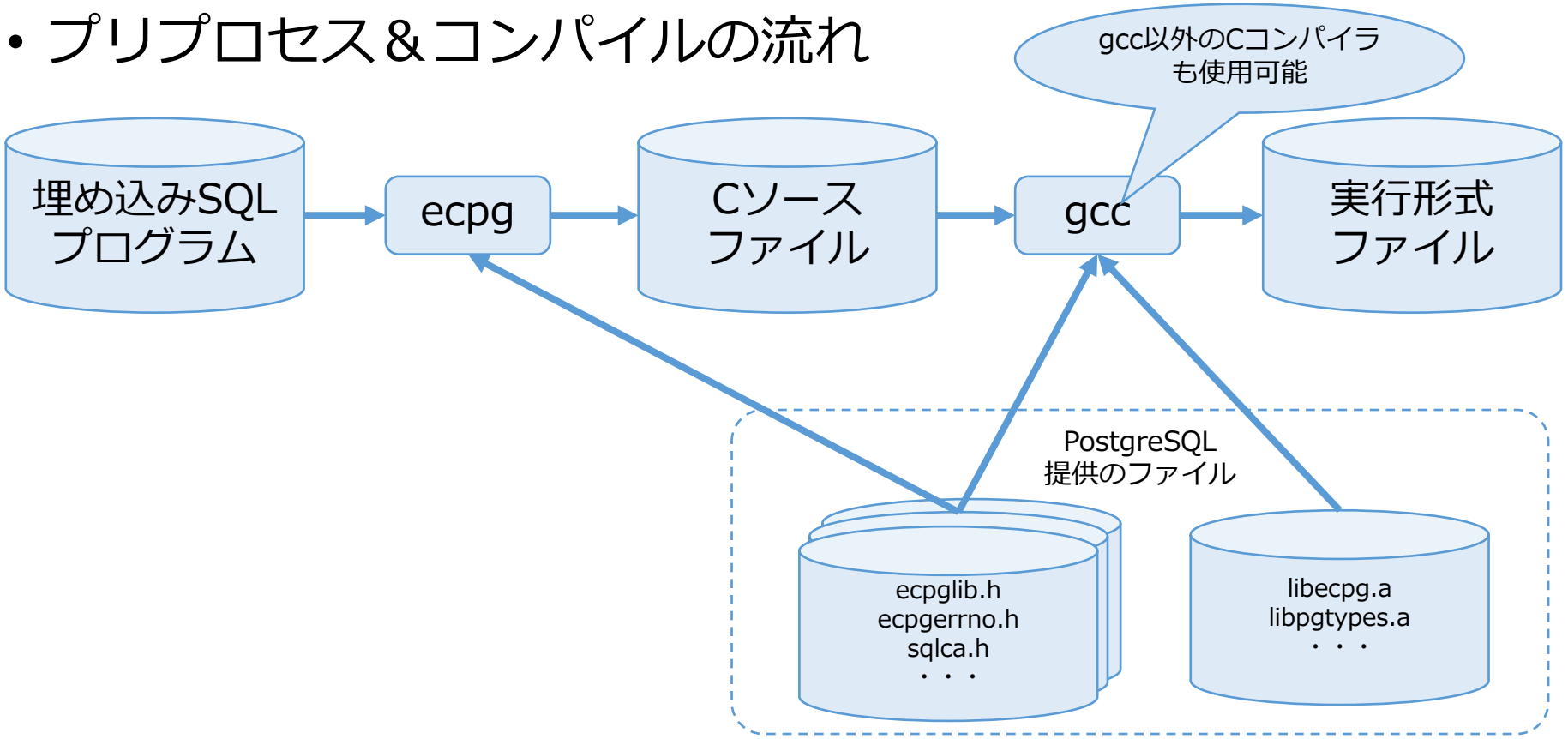
- 標準SQL(SQL:2023)で規定されている埋め込みSQL言語
  - Ada, COBOL, C, Fortran, MUMPS, Pascal, PL/I
- PostgreSQLで対応しているのはC（C++も一部対応）
- 主要DBMSでの対応状況

DBMS	C/C++	COBOL	その他言語
Oracle	○	○	Fortran, PL/I
SQL Server	○	○	不明
Db2	○	○	不明
MySQL	－	－	－
PostgreSQL	○	△ (*1)	不明

(*1) 一部のPostgreSQL商用製品ではCOBOLプリプロセッサをサポートしているが、PostgreSQL標準配布物には含まれない

ECPG

- C言語埋め込みSQLファイルを処理するプリプロセッサ
- 開発用パッケージに含まれる
  - Redhat系：postgresqlXX-devel-XX.XX-XPGDG.*.rpm に含まれる
- プリプロセス＆コンパイルの流れ



ECPGによるプリプロセス

- 埋め込みSQLプログラムのプリプロセス

```
ecpg sample.pgc
```

- ecpgコマンドでプリプロセスを行う。
- 埋め込みSQLプログラムの拡張子は、通常 .pgc を用いる。
- 詳細は ecpg のリファレンス参照
  - https://www.postgresql.jp/document/16/html/app-ecpg.html

- 生成したCソースのコンパイル/リンク

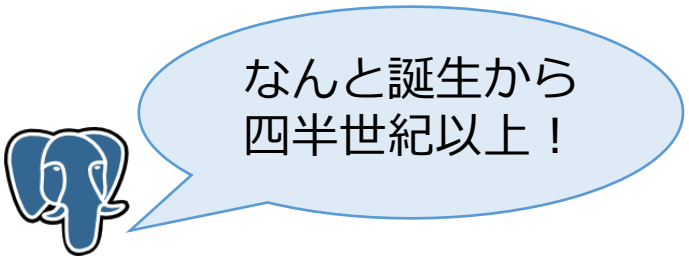
```
cc sample.c -o sample -I ecpg
```

- 環境によってはecpgインクルードファイルの置き場所やecpgライブラリの置き場所を、-Iオプションや-Lオプションで指定する必要がある。
- 使用する機能により -I ecpg 以外のライブラリも指定することもある。

- 大規模な開発ではきちんとmakefileを使う

ECPGヒストリー

バージョン	リリース日	主な変更内容
6.3	1998-03-01	ECPGの誕生
7.0	2000-05-08	指示子の改善、記述子領域のサポート
7.2	2002-02-04	EXECUTE ... INTOのサポート、
7.4	2003-11-17	スレッドセーフ化、Informix互換モードサポート
8.2	2006-12-05	SHOWコマンド対応
8.3	2008-02-04	V3フロントエンド/バックエンドプロトコルの採用
9.0	2010-09-20	SQLDA(SQL記述子領域)のサポート DESCRIBE文のサポート
11	2018-10-18	WHENEVER DO CONTINUEのサポート
12	2019-10-03	bytea型のサポート



埋め込みSQLの書き方

SQLの実行

ホスト変数

指示子

エラー処理（SQLCODE, SQLCA, WHENEVER）

動的SQL

記述子（DESCRIPTOR, SQLDA）

SQLの実行

- EXEC SQL構文
- データベース接続/切断
- DDLの実行
- DMLの実行
- SELECTの実行
- SELECTの実行（カーソル）

EXEC SQL構文

- EXEC SQL構文

```
EXEC SQL 任意のSQL文 ;
```

- SQL文の前に「EXEC SQL」キーワードを入れる。
- SQL文の末尾に「;」（セミコロン）を入れる。
- 複数行にまたがる記述も可能

```
EXEC SQL
  CREATE TABLE IF NOT EXISTS foo
  (id integer primary key, data text)
;
```

接続

- ECPGプログラムでも最初にデータベース接続が必要
- データベースへの接続形式

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

パラメータ	指定要否	意味
target	必須	接続先（ホスト名、ポート番号、データベース名）
connection-name	オプション	接続名
user-name	オプション	ユーザ名

接続（targetの指定）

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

- targetには接続先を指定する。
  - ホスト名、ポート番号、DB名（＋オプション）
- targetのパターン
  - dbname[@hostname][:port]
  - tcp:postgresql://hostname[:port][/dbname][?options]
  - unix:postgresql://localhost[:port][/dbname][?options]
- DB名、ポート番号、ユーザ名はECPGプログラムの実行環境の環境変数として指定することも可能

接続（connection-nameの指定）

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

- 1つのECPGプログラムから1つのデータベースに接続する場合には指定は不要。複数のデータベースに接続するときに指定する。
- SQL実行時にconnection-nameを指定、接続の切替コマンド時に指定、SQL識別子を宣言の3つの使いかたがある。

SQL文実行時にconnection-nameを指定する例

```
EXEC SQL AT connection-name SQL文 ...;
```

接続の切替コマンド例

```
EXEC SQL SET CONNECTION connection-name;
```

接続（user-nameの指定）

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

- user-nameの指定パターン
  - *user-name*
  - *user-name*/*password*
  - *user-name* IDENTIFIED BY *password*
  - *user-name* USING *password*

接続切断

- データベース接続の切断

```
EXEC SQL DISCONNECT [connection];
```

- connectionのバリエーション
  - connection-name
  - CURRENT
  - ALL
- connection-nameの指定がない場合あるいはCURRENTを指定した場合には現在の接続を切断。
- 指定したconnection-nameの接続を切断。
- ALLは全ての接続を切断。

DDLの実行

- EXEC SQLキーワードの後に、DDLのSQL文をそのまま記述すればOK。
- デフォルトでは自動コミットモードになっていない。
  - DISCONNECTする前に、COMMITを忘れずに！

```
int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL
      CREATE TABLE IF NOT EXISTS foo
      (id integer primary key, data text)
    ;

    EXEC SQL DISCONNECT;

    return 0;
}
```

```
int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL
      CREATE TABLE IF NOT EXISTS foo
      (id integer primary key, data text)
    ;

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT;

    return 0;
}
```

- COMMITせずにDISCONNECTすると、実行したSQL文（この例だとCREATE TABLE）はロールバックされる！

DMLの実行

- DML（INSERT, DELETE, UPDATE等）もDDLと同様に記述。

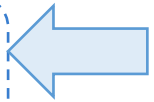
```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

int main() {
    EXEC SQL CONNECT TO mydb USER myuser;
    printf("connected database.¥n");

    EXEC SQL INSERT INTO foo VALUES (1, '鈴木'),(2, '佐藤'),(3,'田中');
    printf("insert table¥n");

    EXEC SQL UPDATE foo SET data = '原田' WHERE id = 2;
    printf("update table¥n");

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT;
    printf("disconnected database.¥n");

    return 0;
}
```

- DMLが何件処理した等の情報の取得方法は後述。

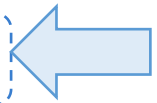
SELECTの実行

- SELECT文の場合、実行結果をCプログラムに引き渡す必要がある。
- 「ホスト変数」を経由して検索結果をCプログラムに引き渡す。
- 単一の行を返却する場合、SELECT文のINTO句にホスト変数を指定する。

```
int main() {
    EXEC SQL CONNECT TO mydb USER myuser;
    printf("connected database.¥n");

    EXEC SQL INSERT INTO foo VALUES (1, '鈴木'),(2, '佐藤'),(3,'田中');
    printf("insert table¥n");

    EXEC SQL BEGIN DECLARE SECTION;
    int cnt = -1;
    int min = -1;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT COUNT(*), MIN(id) INTO :cnt, :min FROM foo;
    printf("cnt = %d, min = %d¥n", cnt, min);

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT;
    printf("disconnected database.¥n");

    return 0;
}
```



ホスト変数を宣言



SELECTの結果を
ホスト変数に格納

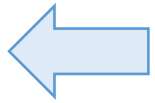
SELECTの実行（カーソルの使用）

- SELECTから複数行の結果を取得するときには「カーソル」を使用する。
  - DECLARE カーソル名 CURSOR FOR SELECT文
  - OPEN カーソル名
  - （ループしながら）FETCH カーソル名 INTO ホスト変数リスト
  - CLOSE カーソル名

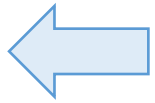
SELECTの実行（カーソルの使用）

- カーソルを使った検索の例

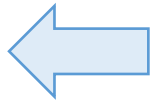
```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

int main() {
    EXEC SQL CONNECT TO mydb USER myuser;
    printf("connected database.¥n");

    EXEC SQL BEGIN DECLARE SECTION;
    int id = -1;
    char data[32];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL DECLARE foo_cur CURSOR FOR SELECT id, data FROM foo;
    EXEC SQL OPEN foo_cur;

    while (1) {
        EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :data ;
        if (sqlca.sqlcode == ECPG_NOT_FOUND)
            break;
        printf("id = %d, data = %s¥n", id, data);
    }

    EXEC SQL CLOSE foo_cur;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;
    printf("disconnected database.¥n");
    return 0;
}
```



カーソル定義とオープン



カーソルから値取得



カーソルをクローズ

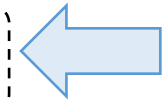
ホスト変数

- Cプログラムと埋め込みSQL文の間で情報をやりとりする仕組み
- 「ホスト」とは？
  - SQLが「ホスト言語」のCプログラムに対する「ゲスト」である
  - なのでCプログラムの変数を「ホスト変数」と呼ぶ
- ホスト変数の基本
  - 「宣言セクション」内で定義
  - 変数名の前に「コロン」を付与するとホスト変数の参照となる

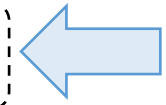
ホスト変数（宣言セクション）

- EXEC SQL BEGIN DECLARE SECTIONと、EXEC SQL END DECLARE SECTIONの間の行で変数定義を行う。
- 宣言セクションは複数記述可能。
- 宣言セクションはグローバル・ローカルどちらでも記述可能。

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

EXEC SQL BEGIN DECLARE SECTION;
  int  in_id;
EXEC SQL END DECLARE SECTION;

int main() {

    EXEC SQL BEGIN DECLARE SECTION;
    char in_data[32];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO mydb USER myuser;
```



グローバル変数として宣言

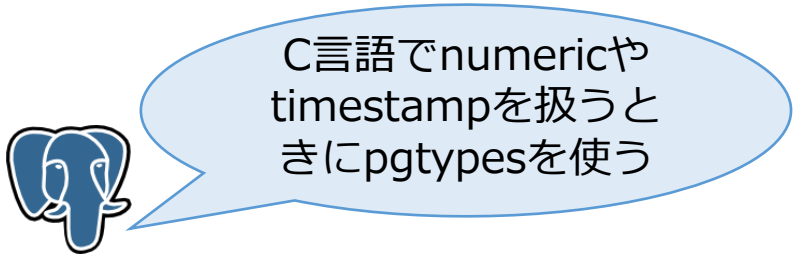


ローカル変数として宣言

ホスト変数（データ型の対応）

- 主なPostgreSQLデータとC言語でのホスト変数型の対応

大分類	PostgreSQLデータ型	ホスト変数宣言時の型	備考
数値型	smallint/integer/bigint	short/int/long long int	
	decimal/numeric	decimal/numeric	pgtypes使用
	real/double precision	float/double	
文字型	char(n)/varchar(n)/text	char[n+1], VARCHAR[*n*+1]	
日付型	timestamp/date	timestamp/date	pgtypes使用
バイナリ型	bytea	char *, bytea[n]	



ホスト変数（単純な整数型、文字列型）

- 整数型と文字列型を使って入出力する

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

EXEC SQL BEGIN DECLARE SECTION;
  int  in_id;
  char in_data[32];
  int  out_id;
  char out_data[32];
EXEC SQL END DECLARE SECTION;

int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL TRUNCATE foo;

    /* ホスト変数に値を設定して挿入 */
    in_id = 1;
    strcpy(in_data, "鈴木");
    EXEC SQL INSERT INTO foo VALUES (:in_id, :in_data);

    printf("insert table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);

    /* 検索結果をホスト変数に設定 */
    out_id = 0;
    strcpy(out_data, ""); // 空値に設定
    printf("select before, out_id = %d, out_data = %s¥n", out_id, out_data);
    EXEC SQL SELECT id, data INTO :out_id, :out_data FROM foo WHERE id = :in_id;
    printf("select after , out_id = %d, out_data = %s¥n", out_id, out_data);

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT;

    return 0;
}
```

- 実行結果

```
$ ./sample
insert table, sqlca.sqlcode = 0
select before, out_id = 0, out_data =
select after , out_id = 1, out_data = 鈴木
$
```

C言語上で値の代入を行う

SQLに渡すときにはコロンを変数名の前につける

ホスト変数（VARCHAR）

- 可変長文字列型（varchar, text）は VARCHAR という型も使用可

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

EXEC SQL BEGIN DECLARE SECTION;
  int  in_id;
  VARCHAR in_data[32];
  int  out_id;
  VARCHAR out_data[32];
EXEC SQL END DECLARE SECTION;

int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL TRUNCATE foo;
    /* ホスト変数に値を設定して挿入 */
    in_id = 1;
    in_data.len = strlen("鈴木");
    strcpy(in_data.arr, "鈴木");
    EXEC SQL INSERT INTO foo VALUES (:in_id, :in_data);
    printf("insert table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);

    /* 検索結果をホスト変数に設定 */
    out_id = 0;
    strcpy(out_data.arr, ""); // 空値に設定
    printf("select before, out_id = %d, out_data.len = %d, out_data.arr = %s¥n", out_id,
out_data.len, out_data.arr);
    EXEC SQL SELECT id, data INTO :out_id, :out_data FROM foo WHERE id = :in_id;
    printf("select before, out_id = %d, out_data.len = %d, out_data.arr = %s¥n", out_id,
out_data.len, out_data.arr);

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT;

    return 0;
}
```

- 実行結果

```
$ ./sample
insert table, sqlca.sqlcode = 0
select before, out_id = 0, out_data.len = 0,
out_data.arr =
select before, out_id = 1, out_data.len = 6,
out_data.arr = 鈴木
$
```

VARCHAR型で宣言した変数は、
以下のC言語構造体に変換される。

```
struct varchar_1  { int len; char arr[ 32 ]; }  in_data ;
```

len には文字列長（終端¥0を含まない長さ）を設定。
arr には文字列本体を設定。

検索結果も同様に格納される。
文字列自体はarrを参照する。

ホスト変数（pgtypesライブラリ）

- C言語にはtimstamp, date, interval, numeric, decimal型を直接表現可能なプリミティブ型はない。
- ECPGでtimestamp，interval，numericを格納・演算するためには、pgtypesライブラリが提供する機能を使う。
- pgtypesライブラリ使用時
  - ECPGプログラムに、使用するデータ型に対応するpgtypes用のヘッダファイルをincludeする。
  - Cソースをビルドするときに、libpgtypes.a をリンクする。
    （ -lpgtypes を指定する。）

ホスト変数（timestamp/interval その1）

- timestamp型を挿入する

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <pgtypes_timestamp.h> /* timestamp用のヘッダファイル */


int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

EXEC SQL BEGIN DECLARE SECTION;
  int  id;
  timestamp t_data; /* 実体として宣言する */
EXEC SQL END DECLARE SECTION;

    EXEC SQL TRUNCATE foo;
    /* ホスト変数に値を設定して挿入 */
    id = 1;
    /* 文字列形式からの変換 */
    t_data = PGTYPEStimestamp_from_asc("2024-12-05 22:00:00", NULL);

    EXEC SQL INSERT INTO foo VALUES (:id, :t_data);
    printf("insert table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);

    EXEC SQL COMMIT;
```

次スライドに続く

timestamp型のホスト変数を宣言する。

文字列からtimestampへの変換を行う。

timestamp型ホスト変数を使ってINSERTする。

- 実行結果

```
$ ./sample
insert table, sqlca.sqlcode = 0
```

ホスト変数（timestamp/interval その2）

- timestamp型を検索しinterval型を使って演算する

```
    EXEC SQL SELECT id, t_data INTO :id, :t_data FROM foo;
    printf("select table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);
    /* 検索したホスト変数にinteval型を加算し結果を文字列で取得 */
    {
        interval *i_data;
        timestamp result;
        char result_text[16];

        i_data = PGTYPESinterval_from_asc("12 hour", NULL);
        /* timestamp型にintarval型を加算する */
        PGTYPEStimestamp_add_interval(&t_data, i_data, &result);
        /* 加算結果をm/d/y HHMMSS形式の文字列として取得する */
        PGTYPEStimestamp_fmt_asc(&result, result_text, 16, "%m/%d/%y %H%M%S");
        printf("result_text(%%m/%%d/%%y %%H%%M%%S) = %s¥n", result_text);

        /* 各領域の解放 */
        PGTYPESinterval_free(i_data);
    }

    EXEC SQL DISCONNECT;

    return 0;
}
```

- 実行結果

```
select table, sqlca.sqlcode = 0
result_text(%m/%d/%y %H%M%S) = 12/06/24 100000
```

2024-12-05 22:00:00
の12時間後を
m/d/y HHMMSS形式で
出力

interval型の変数、timestamp型の変数、文字列用の変数を宣言する。
SQLに渡さないのであれば、ホスト変数宣言ではなく、通常のCの変数宣言でもOK。

検索結果（2024-12-05 22:00:00に、'12 hour'を加算し、その結果を文字列（m/d/y HHMMSS形式）で取得する。

interval型は動的に領域確保しているので解放する必要がある。

ホスト変数（numeric その1）

- numeric型を挿入する

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <pgtypes_numeric.h> /* numeric用のヘッダファイル */


int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

EXEC SQL BEGIN DECLARE SECTION;
  int  id;
  numeric *n_data; /* ポインタとして宣言する */
EXEC SQL END DECLARE SECTION;

    EXEC SQL TRUNCATE foo;
    /* ホスト変数に値を設定して挿入 */
    id = 1;
    /* 文字列形式からnumericへの変換 */
    n_data = PGTYPESnumeric_from_asc("123.4", NULL);

    EXEC SQL INSERT INTO foo VALUES (:id, :n_data);
    printf("insert table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);

    EXEC SQL COMMIT;
```

次スライドに続く

numeric型のホスト変数を宣言する。

文字列からnumericへの変換を行う。
（n_dataは後で解放しなければならない）

numeric型ホスト変数を使ってINSERTする。

- 実行結果

```
$ ./sample
insert table, sqlca.sqlcode = 0
```

ホスト変数（numeric その2）

- 検索したnumericホスト変数に加算して文字列として取得

```
    EXEC SQL SELECT id, n_data INTO :id, :n_data FROM foo;
    printf("select table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);
    /* 検索したホスト変数に数値を加算し結果を文字列で取得 */
    {
        numeric *add_data;
        numeric *result;
        char* result_text;

        add_data = PGTYPESnumeric_new();
        add_data = PGTYPESnumeric_from_asc("0.056", NULL);
        result = PGTYPESnumeric_new();
        /* numeric型を加算する */
        PGTYPESnumeric_add(n_data, add_data, result);
        /* 加算結果を精度6の数字文字列として取得する */
        result_text = PGTYPESnumeric_to_asc(result, 6);
        printf("result_text = %s¥n", result_text);

        /* 各領域の解放 */
        PGTYPESnumeric_free(add_data);
        PGTYPESnumeric_free(result);
        PGTYPESchar_free(result_text);
    }
    /* numericホスト変数の解放 */
    PGTYPESnumeric_free(n_data);

    EXEC SQL DISCONNECT;

    return 0;
}
```

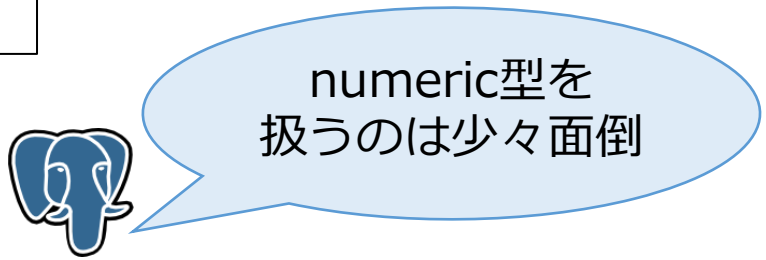
- 実行結果

```
result_text = 123.456000
```

numeric型の変数、文字列用の変数を宣言する。
SQLに渡さないのであれば、ホスト変数宣言ではなく、通常のCの変数宣言でもOK。

検索結果（123.4）に、0.056を加算し、その結果を文字列（精度6）で取得する。

numeric型用の関数で確保した領域を解放する。
（これをやらないとメモリリークの原因になる）



ホスト変数（numeric/timestampの簡易版 その1）

- numericやtimestampの演算をするなら、pgtypesの型や関数を使う必要がある。
- しかしECPGプログラム内で演算をしないなら、文字列型ホスト変数で入出力する手もある。

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

int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

EXEC SQL BEGIN DECLARE SECTION;
  int  id;
  char n_data[32]; /* 数字文字列格納領域 */
  char t_data[32]; /* timestamp文字列格納領域 */
EXEC SQL END DECLARE SECTION;

    EXEC SQL TRUNCATE foo;
    /* ホスト変数に値を設定して挿入 */
    id = 1;
    /* 文字型ホスト変数に数字やtimestampを設定する */
    strcpy(n_data, "123.4");
    strcpy(t_data, "2024-12-06 10:00:00");

    EXEC SQL INSERT INTO foo VALUES (:id, :n_data, :t_data);
    printf("insert table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);

    EXEC SQL COMMIT;
```

文字型のホスト変数を宣言する。

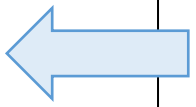
文字列として数字やtimestampを設定する。（書式のチェックはPostgreSQLサーバ側に任せる、という方法）

次スライドに続く

ホスト変数（numeric/timestampの簡易版 その2）

- PostgreSQLからの検索結果を文字型ホスト変数に格納し、そのまま出力

```
    /* 文字型ホスト変数をクリアする */
    strcpy(n_data, "");
    strcpy(t_data, "");

    EXEC SQL SELECT id, n_data, t_data INTO :id, :n_data, :t_data FROM foo;
    printf("select table, sqlca.sqlcode = %d¥n", sqlca.sqlcode);
    printf("id = %d, n_data=%s, t_data=%s¥n", id, n_data, t_data);

    EXEC SQL DISCONNECT;

    return 0;
}
```



numeric型の列もtimestamp型の列も文字型ホスト変数に格納される。
格納先の文字型ホスト変数がオーバーフローしないようにホスト変数は宣言する。
timestampの書式は環境に依存することに注意が必要

- この方式はJSON型やユーザ定義型等をECPGで扱う場合にも使える。
  - PostgreSQLの特殊な型（JSON型等）や、ユーザ定義型をECPGアプリケーションで扱うのは難しい・・・。
  - 例えばJSONであれば、なるべくSQL内でJSON操作を行い、アプリケーションには値のみを返却する等の工夫が必要。

指示子

- RDBMSにはNULLという概念がある。
- 空文字≒NULL
  - Oracleでは空文字とNULLを区別しないが、PostgreSQLでは厳密に区別をしている。
  - PostgreSQLでは列値に空文字を設定してもNULLにはならない。
  - そもそも数値型のNULLってどう表現する？
- ECPGでは、ホスト変数とは別の「指示子」と呼ばれる変数を使ってNULLを扱う。
- 指示子はホスト変数宣言セクション内で、int型で定義する。
- ホスト変数の直後に指示子を記述する。
- 指示子の値が-1なら列値をNULLとみなす。

指示子（データ挿入時）

- 指示子に-1を設定してNULL値を挿入する。

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

int main() {
    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL BEGIN DECLARE SECTION;
    int id ;
    int i_data;
    char t_data[32];

    int i_ind;  /* i_data用の指示子 */
    int t_ind;  /* t_data用の指示子 */

    EXEC SQL END DECLARE SECTION;

    EXEC SQL TRUNCATE foo;

    /* データ挿入時にNULLを設定する */
    /* id=1のi_dataはNULL */
    id = 1;
    strcpy(t_data, "ABCDEFG");

    i_ind = -1, t_ind = 0;

    EXEC SQL INSERT INTO foo VALUES (:id, :i_data :i_ind, :t_data :t_ind);

    /* id=2のt_dataはNULL */
    id = 2;
    i_data = 100;
    strcpy(t_data, "ABCDEFG"); // 無視される
    i_ind = 0, t_ind = -1;
    EXEC SQL INSERT INTO foo VALUES (:id, :i_data :i_ind, :t_data :t_ind);

    EXEC SQL COMMIT;
```

ホスト宣言セクションで指示子の変数を宣言する。

ホスト変数をNULLにしたいときは、指示子（i_ind）に -1 を設定する。

SQL内でホスト変数を使うときに、
:ホスト変数名:指示子
という形式で記述する。
（ホスト変数名と指示子の間の空白はあってもなくても良い）

次スライドに続く

指示子（データ検索時）

- 指示子を確認して、-1ならNULLと判断する。

```
    EXEC SQL DECLARE foo_cur CURSOR FOR SELECT id, i_data, t_data FROM foo;
    EXEC SQL OPEN foo_cur;

    while (1) {
        EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :i_data:i_ind, :t_data:t_ind ;

        if (sqlca.sqlcode == ECPG_NOT_FOUND || sqlca.sqlcode < 0)
            break;

        printf("id=%d, ", id);

        /* i_dataの出力 */
        if (i_ind == -1) {
          printf("i_data is null, ");
        } else {
          printf("i_data=%d, ", i_data);
        }

        /* t_dataの出力 */
        if (t_ind == -1) {
          printf("t_data is null¥n");
        } else {
          printf("t_data=%s¥n", t_data);
        }
    }

    EXEC SQL CLOSE foo_cur;
    EXEC SQL DISCONNECT;

    return 0;
}
```

- 実行結果

```
$ ./sample
id=1, i_data is null, t_data=ABCDEFG
id=2, i_data=100, t_data is null
$
```

SQL内でホスト変数を使うときに、
:ホスト変数名:指示子
という形式で記述する。

ホスト変数の参照の前に、指示子の値を確認して、
-1 であればホスト変数値がNULLと判断する。

エラー処理

- ECPGプログラムに記述したDB接続処理、SQL実行処理で何らかのエラーが発生する可能性がある。
- ECPGプログラムでのエラー処理の扱い
  - SQLCODEとSQLSTATE
  - SQLCA
  - WHENEVER句

エラー処理（SQLCODEとSQLSTATE）

- ECPGでは以下の2種類のエラーコードを参照可能

	SQLCODE	SQLSTATE
データ型	integer	char[5]
SQL標準	非準拠（SQL-92版で廃止）	準拠
正常	0	00000
準正常系（0件、カーソル末端）	100 (*1)	02000
エラー	負の値（PostgreSQL文書 SQLSTATE対SQLCODE 参照）	（PostgreSQL文書 付録A PostgreSQLエラーコード 参照）

(*1) Oracle Pro*Cでは1403 が設定される。

- 処理判定はSQLCODEのほうが楽。
- エラー内容の詳細度はSQLSTATEのほうが高い。
- 後述のWHENEVERではSQLCODEを参照している。
- pgypeslib固有のエラー種別はSQLSTATEではなく、SQLCODEで判別が必要。

エラー処理（SQLCA）

- SQLCA（SQL Communication Area）
  - SQL実行後の状態・エラー詳細が格納される構造体。
  - 前述のSQLCODEやSQLSTATEも格納されている。
  - SQL実行時に複数の警告やエラーが発生した場合、SQLCAは最後の情報のみ格納される。
- SQLCAの利用方法
  - ECPGプログラム上での宣言等は不要。
  - エラー処理を行う部分で "sqlca.メンバ変数" を書く。
  - ecpgコマンドでプリコンパイルするとSQLCA領域をグローバル変数としてCソースに展開する。
    - マルチスレッド時、各スレッドにSQLCAのコピーを生成する。

エラー処理（SQLCA）

- SQLCAの内容

メンバ変数名・データ型	意味	用途
char sqlcaid[8]	未実装	用途なし
long sqlabc	未実装	用途なし
long sqlcode	SQLCODE	エラー判定
struct sqlerrm	エラーメッセージ構造体	
int sqlerrml	エラーメッセージ長	エラーメッセージ取得
char sqlerrmc[SQLERRMC_LEN];	エラーメッセージ	エラーメッセージ取得
char sqlerrp[8]	未実装	用途なし
long sqlerrd[6]	[1]：処理された行のOID [2]：処理された件数	
char sqlwarn[8]	警告状態フラグ	警告の詳細な判別
char sqlstate[5]	SQLSTATE	エラー情報表示/エラー判定

エラー処理（WHENEVER）

- SQL発行毎に前述のSQLCODEやSQLSTATEを使ったエラー判定やエラー時の処理を書くのは面倒。
  - コードの可読性も落ちる。
- WHENEVER文を使うとエラー処理記述を簡易化できる。

```
EXEC SQL WHENEVER condition action;
```

- WHENVERはcondtionとactionを持つ
  - condtion：条件を定義
  - action：条件を満たしたときに行う動作
- conditionとactionの詳細は次スライド参照

エラー処理（WHENEVER）

- conditionとaction

種別	指定文字列	意味
condition	SQLERROR	SQLエラー発生時
	SQLWARN	警告発生時
	NOT FOUND	0件を受け取る、または0件に影響（更新等）
action	CONTINUE	実質的にはconditionの無視
	GO TO *Label* GOTO *Label*	GOTO文で指定Labelへジャンプ
	SQLPRINT	簡易なメッセージ出力
	STOP	プログラムの停止。exit()を呼び出す。
	DO BREAK	Cループから脱出
	DO CONTINUE	Cループ先頭に戻る
	CALL *name* (*args*) DO *name* (*args*)	C関数の呼び出し

エラー処理（WHENEVER）

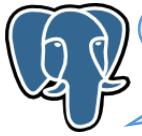
- WHENEVERの典型的な使い方
  - SQLERROR発生時はエラー処理を行って処理中断
  - 0件取得（カーソル末端到達）時はループ脱出
- WHENEVERを使うことで、SQL実行後に毎回判定や、エラー処理の記述が不要になる。

WHENEVERを使わない場合

```
EXEC SQL CONNECT TO mydb USER myuser;
if ( sqlca.sqlcode < 0 ) {
    /* エラー処理 */
}

EXEC SQL DECLARE foo_cur CURSOR FOR SELECT id, data FROM foo;
EXEC SQL OPEN foo_cur;
if ( sqlca.sqlcode < 0 ) {
    /* エラー処理 */
}

while (1) {
    EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :data ;
    if ( sqlca.sqlcode < 0 ) {
        /* エラー処理 */
    }
    if ( sqlca.sqlcode == ECPG_NOT_FOUND ) {
        /* ループ脱出処理 */
    }
    printf("id = %d, data = %s¥n", id, data);
}
```

WHENEVERを使う場合

```
EXEC SQL WHENEVER SQLERROR GOTO ERROR_LABEL;
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL CONNECT TO mydb USER myuser;

EXEC SQL DECLARE foo_cur CURSOR FOR SELECT id, data FROM foo;
EXEC SQL OPEN foo_cur;

while (1) {
    EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :data ;
    printf("id = %d, data = %s¥n", id, data);
}
```



エラー処理の記述や
ループ脱出判定の記述
がシンプルに！

エラー処理（WHENEVER）

- WHENEVERの使用例

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

int main() {
    EXEC SQL BEGIN DECLARE SECTION;
    int id = -1;
    char data[32];
    EXEC SQL END DECLARE SECTION;

    /* WHENEVER指定 */
    EXEC SQL WHENEVER SQLERROR GOTO ERROR_LABEL;
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL DECLARE foo_cur CURSOR FOR SELECT id, data FROM foo;
    EXEC SQL OPEN foo_cur;

    while (1) {
        EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :data ;
        printf("id = %d, data = %s¥n", id, data);
    }

    EXEC SQL CLOSE foo_cur;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;

    return 0; /* 正常終了 */

ERROR_LABEL:
   printf("SQL Error, SQLSTATE=%5s¥nsqlerrmc=[%s]¥n",
       sqlca.sqlstate, sqlca.sqlerrm.sqlerrmc);
   return -1; /* 異常終了 */
}
```

- 実行結果（異常時）

```
$ ./sample
id = 1, data = ABC
id = 2, data = DEF
id = 3, data = XYZ
$
```

- 実行結果（異常時）

```
$ ./sample
SQL Error, SQLSTATE=42P01
sqlerrmc=[relation "foo" does not exist on line 18]
$
```

SQLERROR時に、"ERROR_LABLE"へジャンプする。

NOT FOUND（クエリ結果0件、またはカーソル末端到達時）はwhileループから脱出する。

SQLERROR時のエラー処理

動的SQL

- ここまでの説明ではECPGソース内にSQL文を埋め込んでいた。
  - このSQL文はプリコンパイル時に解釈される
- しかしアプリケーションの要件によっては、プリコンパイル時に実行するSQL文が確定しないケースもある。
  - SQL文をプログラム内で組み立てるケース
  - SQL文を外部ソース（ファイル等）から取得するケース
- ECPGでの動的SQLの実行方法
  - EXECUTEでの実行
  - プリペアド文

動的SQL（EXECUTE/プリペアド文）

- SQL文字列をEXECUTE/プリペアド文で処理

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

int main() {
    EXEC SQL BEGIN DECLARE SECTION;
      int id = 0;
      char data [16];
      char *sql = "TRUNCATE foo;";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO mydb USER myuser;

    EXEC SQL EXECUTE IMMEDIATE :sql;

    EXEC SQL PREPARE s1 FROM "INSERT INTO foo VALUES (?, ?)";
    EXEC SQL EXECUTE s1(1, '鈴木');
    EXEC SQL EXECUTE s1(2, '佐藤');
    EXEC SQL EXECUTE s1(3, '田中');

    EXEC SQL PREPARE s2 FROM "SELECT id, data FROM foo WHERE id = ?";
    EXEC SQL EXECUTE s2 INTO :id, :data USING 2;

    printf("id = %d, data = %s¥n", id, data);

    /* プリペアド文を解放する */
    EXEC SQL DEALLOCATE PREPARE s1;
    EXEC SQL DEALLOCATE PREPARE s2;

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;
    return 0;
}
```

TRUNCATEを実行

INSERT文のプリペアド文を作成してEXECUTE時にパラメータを渡して実行する。

SEELCTでも同様に処理可能。

記述子

- レコードのメタデータを管理するために、ECPGでは以下の2つの方法を提供する。
  - 名前付き記述子領域
  - SQLDA記述子領域

記述子（名前付き記述子領域）

- 名前付き記述子領域は、ECPGが提供する埋め込みSQLによって生成する領域である。
  - 名前付き記述子領域は、メタデータも含むホスト変数の役割をもつ
  - 記述子全体のヘッダ情報（現状は列数のみ）
  - 列に対応する項目記述子領域（データとメタデータ）
- 名前付き記述子領域のおおまかな使い方
  - ALLOCATE DESCRIPTOR で領域を確保
  - DESCRIBEで結果セットのメタデータを取得
  - FETCH等の結果をINTOで記述子領域に設定
  - GET DESCRIPTORで記述子領域からメタデータを取得
  - DEALLOCATE DESCRIPTORで領域を解放

記述子（取得できる情報）

- 記述子から取得可能な主な情報を示す。

項目名	データ型	内容
CARDINALITY	整数型	結果セットの行数
DATA	不定	データ項目
INDICATOR	整数型	指示子（NULL値、値の切り詰めを示す）
NAME	文字型	列名
RETURNED_LENGTH	整数型	結果の文字数
RETURNED_OCTET_LENGTH	整数型	結果のバイト数
TYPE	整数型	型を示す整数値

- DATAを取得する場合は、事前にTYPEを確認し、TYPEに応じたホスト変数に格納する。
- TYPEの詳細は次スライド参照

記述子（TYPE用の定数）

- 記述子から取得したTYPEを判定する場合には、sql3type.hで定義したenumを使う。
- 以下に主なデータ型の定数を示す。

enum名	値	備考
SQL3_CHARACTER	1	固定長文字列型
SQL3_NUMERIC	2	
SQL3_DECIMAL	3	
SQL3_INTEGER	4	
SQL3_SMALLINT	5	
SQL3_REAL	7	
SQL3_DOUBLE_PRECISION	8	
SQL3_DATE_TIME_TIMESTAMP	9	更に区別したい場合には、datetime_interval_codeを取得して判断する
SQL3_INTERVAL	10	
SQL3_CHARACTER_VARYING	12	可変長文字列型

- sql3type.h を使う場合は、はC言語の#include ではなく、EXEC SQL INCLUDE文を使う。

```
EXEC SQL INCLUDE sql3types;
```

【参考】sql3type.h は、(PostgreSQLインストール先)/include 配下にある。

記述子（名前付き記述子からの取得フロー）

- 複数行・複数列を取得する検索結果を、名前付き記述子領域からの取得する場合の疑似コード

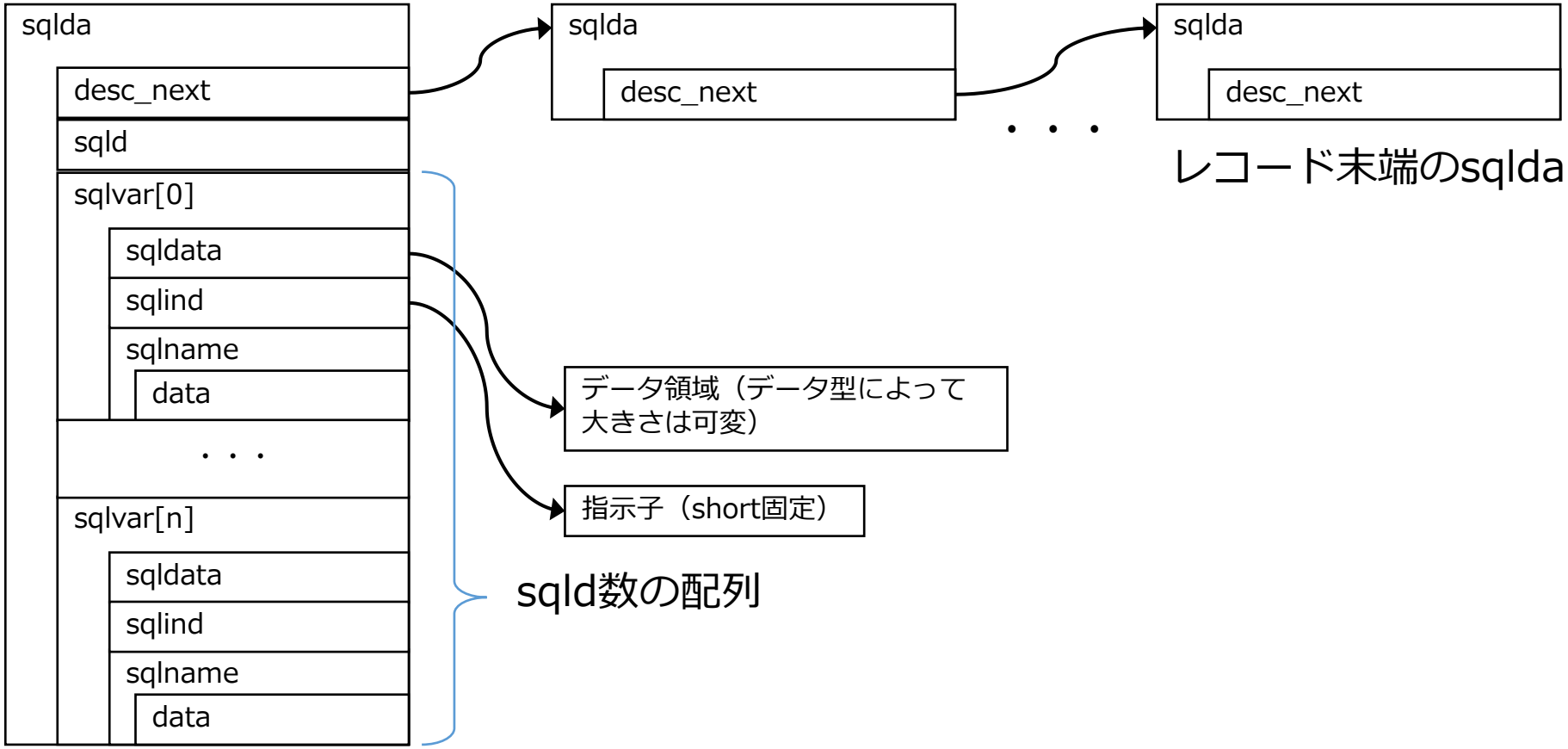
```
EXEC SQL ALLOCATE DESCRIPTOR 記述子名; /* 記述子領域descを確保 */

EXEC SQL DECLARE カーソル名 CURSOR FOR SQL文;
EXEC SQL OPEN カーソル名;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
while (1) {
    EXEC SQL FETCH NEXT FROM カーソル名 INTO SQL DESCRIPTOR 記述子名;
    /* 列数を取得する */
    EXEC SQL GET DESCRIPTOR 記述子名 :列数用ホスト変数 = COUNT;

    for ( 列数分ループする条件 ) {
        /* メタデータを取得する */
        EXEC SQL GET DESCRIPTOR 記述子名 VALUE ・・・

        /* データ取得時には先にTYPEを取得 */
        EXEC SQL GET DESCRIPTOR mydesc VALUE :列番号 :型用ホスト変数 = TYPE;
       /* TYPEに応じたホスト変数にDATAの値を格納する。 */
        switch (型用ホスト変数) {
        case SQL3_INTEGER:
        case SQL3_SMALLINT:
          EXEC SQL GET DESCRIPTOR mydesc VALUE :列番号 :値用ホスト変数 = DATA;
          break;
        default:
          EXEC SQL GET DESCRIPTOR mydesc VALUE :列番号 :値用ホスト変数 = DATA;
          break;
        }
    }
}
/* カーソル・記述子を解放する */
EXEC SQL CLOSE カーソル名;
EXEC SQL DEALLOCATE DESCRIPTOR 記述子名;
```

まず列数を取得する

データ(DATA)を取得する場合は、先にデータ型(TYPE)を取得する

SQLDA記述子領域

- SQLDA（SQL Descriptor Area）は問い合わせの結果セットとメタデータを格納するC言語の構造体
- SQLDAは以下のようなリスト構造になっている。



SQLDA記述子領域の使い方

- SQLDA記述子を使って検索する場合の手順概要
  - ①SQLDAを宣言する。
  - ②FETCH/EXECUTE/DESCRIBEを実行してSQLDAに情報を格納する。
  - ③sqlda構造体のメンバsqldから列数を取得する。
  - ④sqlda構造体のメンバsqlvar[0]、・・・、sqlvar[n-1]などから各列の値やメタデータを取得する。n はsqldで取得した値になる。
  - ⑤sqlda構造体のメンバdesc_nextポインタを追い、次の行（sqlda構造体）を参照し、③④の処理を行う。
  - ⑥全ての処理が終わったらSQLDAを解放する。
- SQLDA記述子を使って入力パラメータも設定可能
  - 本スライドでは説明割愛

SQLDA記述子領域サンプル

- 2レコードFETCHしてSQLDAを出力する

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <pgtypes_numeric.h> /* numeric用のヘッダファイル */

EXEC SQL INCLUDE sql3types;
EXEC SQL INCLUDE sqlda;
sqlda_t *mysqlda;

void print_sqlda(sqlda_t *); /* SQLDA出力関数 */

int main() {
    EXEC SQL BEGIN DECLARE SECTION;
      int cnt = -1;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL WHENEVER SQLERROR GOTO ERROR_LABEL;
    EXEC SQL CONNECT TO mydb USER myuser;
    EXEC SQL TRUNCATE foo;
    EXEC SQL INSERT INTO foo VALUES
      (1, 0.334, 'Tigers'),(2, 0.48, 'Giants'),(3, 0.512, 'DeNA') ;
    EXEC SQL COMMIT;
    EXEC SQL DECLARE foo_cur CURSOR FOR SELECT * FROM foo;
    EXEC SQL OPEN foo_cur;
    EXEC SQL WHENEVER NOT FOUND DO BREAK;
    while (1) {
        EXEC SQL FETCH 2 FROM foo_cur INTO DESCRIPTOR mysqlda;
        print_sqlda( mysqlda );
    }
    /* カーソルを解放する */
    EXEC SQL CLOSE foo_cur;
    EXEC SQL DISCONNECT;
    return 0;
ERROR_LABEL:
    printf("SQL ERROR, SQLSTATE=%5s, msg=%s¥n", sqlca.sqlstate, sqlca.sqlerrm.sqlerrmc);
    return -1;
}
```

データを3レコード挿入

2レコードづつFETCHして、
SQLDA内のデータを出力する。
（次スライド参照）

SQLDA記述子領域サンプル

- SQLDAを出力する関数

```
void print_sqlda(sqlda_t *sqlda) {
    sqlda_t *cur_sqlda = sqlda;

    while (cur_sqlda != NULL) {
        /* 1つのsqlda_tを処理 */
        int idx = 0;
        /* 列数分sqlvar配列をアクセス */
        for (idx = 0; idx < cur_sqlda->sqld; idx++) {
            sqlvar_t v = cur_sqlda->sqlvar[idx];
            /* データ型によって処理分岐 */
            switch (v.sqltype) {
                case ECPGt_int: /* 整数型 */
                    printf("sqlvar[%d].sqldata = %d¥n", idx, *(v.sqldata));
                    break;
                case ECPGt_numeric: /* numeric型 */
                    char *num_buf;
                    num_buf = PGTYPESnumeric_to_asc((numeric*)v.sqldata, -1);
                    printf("sqlvar[%d].sqldata = %s¥n", idx, num_buf);
                    PGTYPESchar_free(num_buf);
                    break;
                default: /* textはここで処理 */
                    char c_buf[1024];
                    memset(c_buf, 0, sizeof(c_buf));
                    memcpy(c_buf, v.sqldata, v.sqllen);
                    printf("sqlvar[%d].sqldata = %s¥n", idx, c_buf);
                    break;
            }
        }
        /* 次にsqlda_tへ移動 */
        cur_sqlda = cur_sqlda->desc_next;
    }
}
```

列数分処理を行う

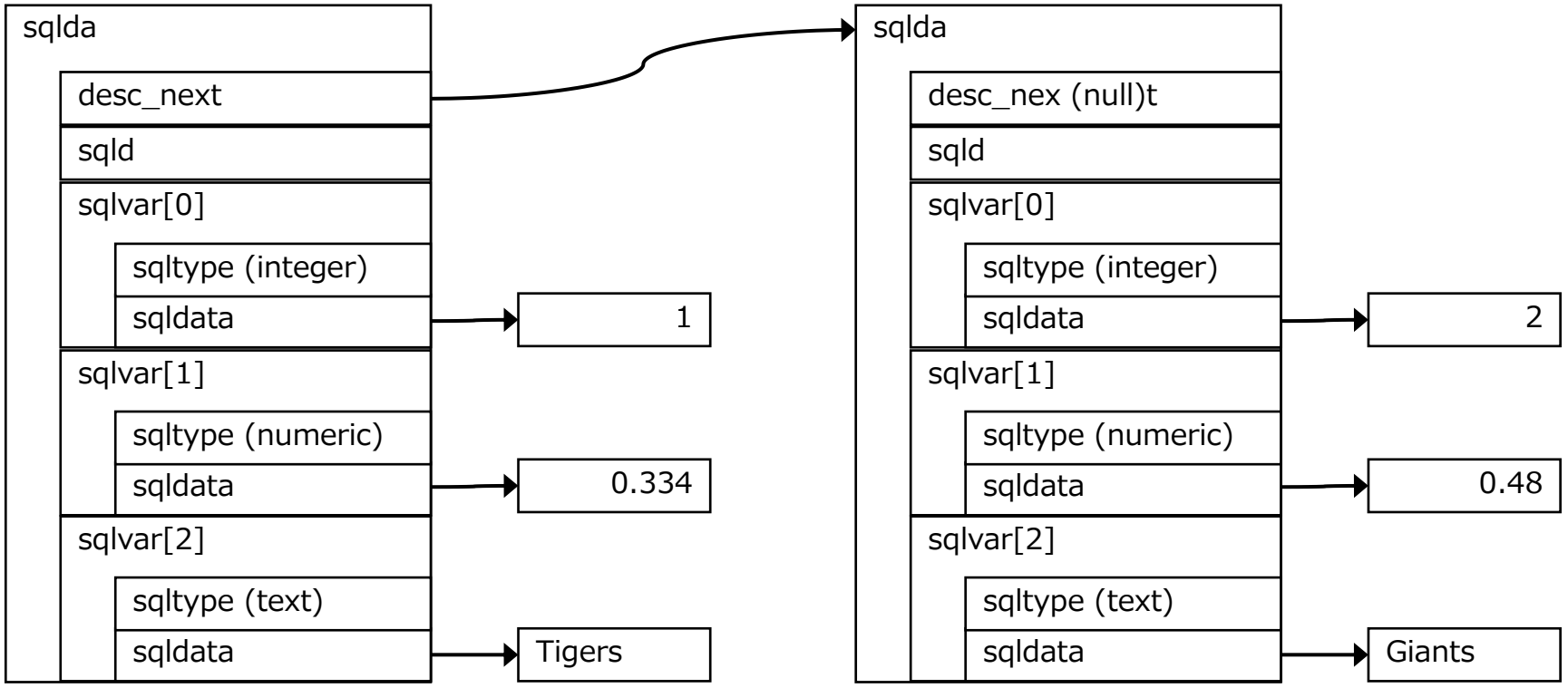
次のSQLDAへ移動

実行結果

```
$ ./sample
sqlvar[0].sqldata = 1
sqlvar[1].sqldata = 0.334
sqlvar[2].sqldata = Tigers
sqlvar[0].sqldata = 2
sqlvar[1].sqldata = 0.48
sqlvar[2].sqldata = Giants
sqlvar[0].sqldata = 3
sqlvar[1].sqldata = 0.512
sqlvar[2].sqldata = DeNA
$
```

SQLDA記述子領域サンプル

- 前スライドの場合、初回のFETCHで取得した2レコードの内容は、以下のようにSQLDA記述子領域に格納されている。



Pro*Cプログラムからの移行

Pro*CからECPGへの移行

共通する部分

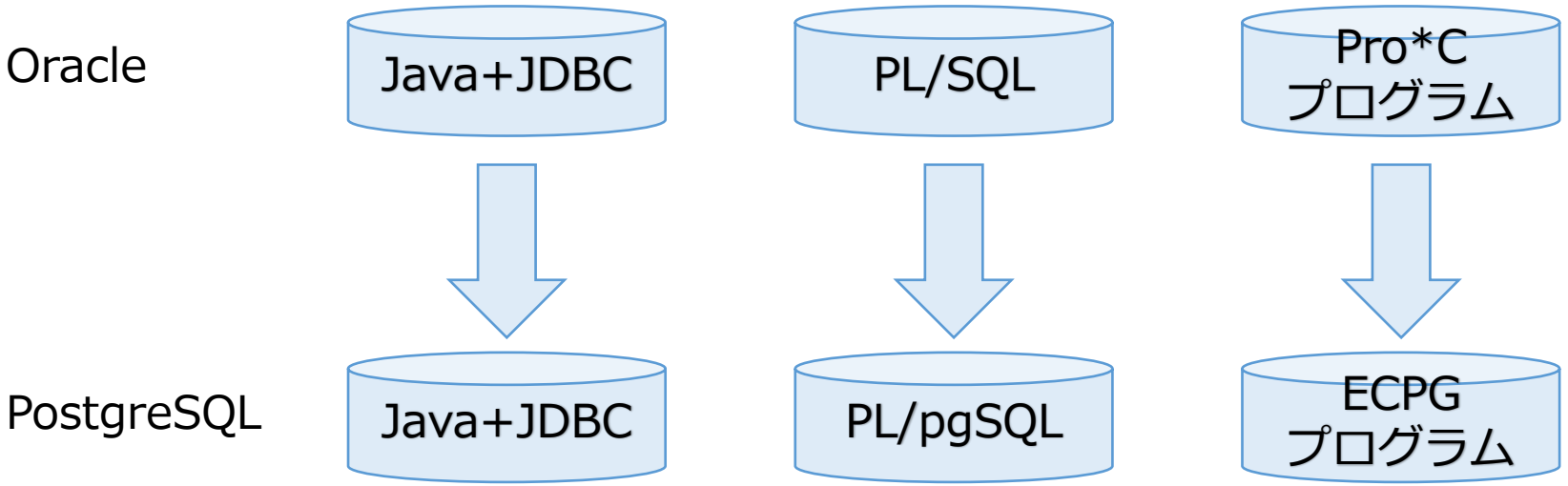
共通しない部分

SQLDAの移行

Oracle互換モード

Pro*CからECPGへの移行

- OracleからPostgreSQLへの移行案件は多い。
- Java＋JDBCで組む案件も多いが、バッチ処理などは、Pro*Cを使っている案件も多い。
- Pro*Cプログラムを移行するときには、まずECPGプログラムへの移行を考える。



Pro*CからECPGへの移行

- シンプル、かつOracle固有の機能をつかっていなければPro*C→ECPG移行は比較的容易。
  - 埋め込みSQL自体の枠組みは使える、C言語のままで移行できる。
- Pro*Cプログラムを無修正でEPCGプログラムとしては使えないと考えたほうがいい。
  - 次スライドのようにいろいろな差異はある。

共通しない部分

- 埋め込みSQLの建前と実態
  - 埋め込みSQLはSQL標準だが、実装依存の機能が多い。
- Pro*CとECPGの差異
  - Oracle/PostgreSQLのSQL文自体の差異。
  - NULLの扱い
  - errorcode
  - SQLDA
  - SQLCA

NULLの扱い

- 文字列に空文字列を設定したときの挙動
  - Oracle：NULLと同じ扱いになる。
  - PostgreSQL：NULLと空文字列は区別される。
- ECPGのOracle互換モード
  - ecpgのオプションに -C ORACLE を指定するとOracle互換モードとしてソースを生成する。
    - 空の文字列型を受け取った場合には、NULL指示子を-1に設定する
    - https://www.postgresql.jp/document/16/html/ecpg-oracle-compat.html

errorcode/SQLSTATE

- errorcodeは標準化されていないので、OracleとPostgreSQLでは異なる。
  - 直前のSQL実行でエラーがなければ0というのは共通。
  - エラーは0より小さい数値、というのは共通。しかし、個々のエラー事象を示すコードは異なる。
  - 特にNOT FOUNDを示す値が異なる
    - Oracle=1403
    - PostgreSQL=100
  - ERRORやNOT FOUNDの判定に直接数値を評価せずに、WHENEVER NOT FOUND/WHENEVER SQLERRORを使っていれば問題ないはず。
- SQLSTATEも（建前上は）SQL標準準拠だが、SQLSTATEを判定している処理は要注意。

SQLDA

- SQLDAはOracleとPostgreSQLで全く異なる！

Oracle

```
struct SQLDA {
  long    N; /* Descriptor size in number of entries        */
  char  **V; /* Ptr to Arr of addresses of main variables    */
  long   *L; /* Ptr to Arr of lengths of buffers             */
  short  *T; /* Ptr to Arr of types of buffers               */
  short **I; /* Ptr to Arr of addresses of indicator vars    */
  long    F; /* Number of variables found by DESCRIBE       */
  char  **S; /* Ptr to Arr of variable name pointers         */
  short  *M; /* Ptr to Arr of max lengths of var. names      */
  short  *C; /* Ptr to Arr of current lengths of var. names  */
  char  **X; /* Ptr to Arr of ind. var. name pointers        */
  short  *Y; /* Ptr to Arr of max lengths of ind. var. names */
  short  *Z; /* Ptr to Arr of cur lengths of ind. var. names */
};
```

- SQLDA情報の構造も異なる。
- Oracle：レコードとカラムの2次元配列で保持
- PostgreSQL：レコード1件ごとにSQLDA構造体が存在し、リスト(sqlda_t.desc_next)で返却される

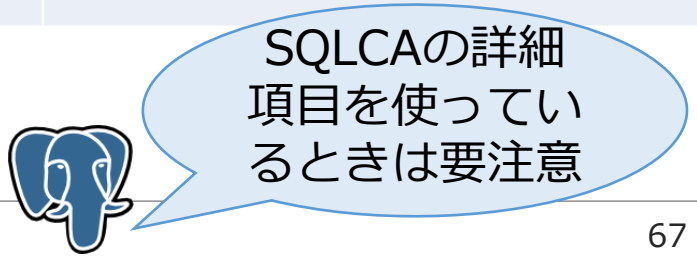
PostgreSQL

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;

struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;

#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

SQLCA

- SQLCAの構造はOracleとPostgreSQ L でほぼ同一。
- しかし一部のメンバ変数の扱いが異なる。以下は一例。

メンバ変数	Oracle	PostgreSQL
sqlerrd[1]	未使用	処理された行のOID
sqlerrd[2]	最後に実行したSQL文によって処理された行数。 処理済行数はOPEN文の後に0(ゼロ)に設定され、FETCH文の後に増分される。	処理された、もしくは返された行数
sqlwarn[0]	別の警告フラグが設定されている	sqlwarn[1]、sqlwarn[2]で示す以外の警告時にWを設定
sqlwarn[1]	ホスト変数への切り詰め発生	ホスト変数への切り詰め発生
sqlwarn[2]	SQLグループ関数の結果にNULL列が使用されない場合	警告時は一律Wを設定



おわりに

ECPGの使いどころ

ECPGの使いどころ

- 移行
  - Oracle→PostgreSQL移行時のPro*Cプログラムの移行
- 新規開発
  - Webアプリ等の開発には向かない
  - バッチ処理をC言語で記述するケース
  - 最初は少しとっつきにくいが、シンプルなSQL処理プログラムを書くのは結構楽かも。
  - ECPGプログラムのひな形を生成AI（ChatGPT等）で作ってみる手もある。

その他

ECPGプログラムのデバッグ

埋め込みC++

ECPGプログラムのデバッグ

- gdbを使う
  - ecpgで生成したCソースをgccの-cオプションでビルドすると、gdb内で、EXEC SQL ...；文を認識してステップ実行が可能。

```
(gdb) b 18
Breakpoint 1 at 0x4011c7: file /home/tosy/ecpg/whenever-01/sample.pgc, line 18.
(gdb) run
Starting program: /home/tosy/ecpg/whenever-01/sample

Breakpoint 1, main () at /home/tosy/ecpg/whenever-01/sample.pgc:18
18          EXEC SQL OPEN foo_cur;
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.34-113.el9.x86_64
(gdb) n
19
(gdb) n
21              EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :data ;
(gdb) n
22              printf("id = %d, data = %s¥n", id, data);
(gdb) n
id = 1, data = Oracle
21              EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :data ;
(gdb) n
22              printf("id = %d, data = %s¥n", id, data);
(gdb) n
id = 2, data = PostgreSQL
21              EXEC SQL FETCH NEXT FROM foo_cur INTO :id, :data ;
(gdb) n
22              printf("id = %d, data = %s¥n", id, data);
(gdb) n
25          EXEC SQL CLOSE foo_cur;
(gdb) n
26          EXEC SQL COMMIT;
(gdb)
```

埋め込みC++

- ECPGはC++に完全には対応していない。
- 安全に使うなら、C++コード部分とCコード部分を分離し、Cコード部分に埋め込みSQL実装をまとめる。