

Tibero

tbESQL/C 안내서

Tibero 7

TMAXTibero

Copyright © 2025 TmaxTibero Co., Ltd. All Rights Reserved.

Copyright Notice

Copyright © 2025 TmaxTibero Co., Ltd. All Rights Reserved.

대한민국 경기도 성남시 분당구 정자일로 45, 티맥스소프트타워 우)13613

Website

<http://www.tmaxtibero.com>

기술서비스센터

Tel : +82-1544-8629

E-Mail : info@tmax.co.kr

Restricted Rights Legend

All TmaxTibero Software (Tibero®) and documents are protected by copyright laws and international convention. TmaxTibero software and documents are made available under the terms of the TmaxTibero License Agreement and this document may only be distributed or copied in accordance with the terms of this agreement. No part of this document may be transmitted, copied, deployed, or reproduced in any form or by any means, electronic, mechanical, or optical, without the prior written consent of TmaxTibero Co., Ltd. Nothing in this software document and agreement constitutes a transfer of intellectual property rights regardless of whether or not such rights are registered) or any rights to TmaxTibero trademarks, logos, or any other brand features.

This document is for information purposes only. The company assumes no direct or indirect responsibilities for the contents of this document, and does not guarantee that the information contained in this document satisfies certain legal or commercial conditions. The information contained in this document is subject to change without prior notice due to product upgrades or updates. The company assumes no liability for any errors in this document.

이 소프트웨어(Tibero®) 사용설명서의 내용과 프로그램은 저작권법과 국제 조약에 의해서 보호받고 있습니다. 사용설명서의 내용과 여기에 설명된 프로그램은 TmaxTibero Co., Ltd.와의 사용권 계약 하에서만 사용이 가능하며, 사용설명서는 사용권 계약의 범위 내에서만 배포 또는 복제할 수 있습니다. 이 사용설명서의 전부 또는 일부분을 TmaxTibero의 사전 서면 동의 없이 전자, 기계, 녹음 등의 수단을 사용하여 전송, 복제, 배포, 2차적 저작물작성 등의 행위를 하여서는 안 됩니다.

이 소프트웨어 사용설명서와 프로그램의 사용권 계약은 어떠한 경우에도 사용설명서 및 프로그램과 관련된 지적 재산권(등록 여부를 불문)을 양도하는 것으로 해석되지 아니하며, 브랜드나 로고, 상표 등을 사용할 권한을 부여하지 않습니다. 사용설명서는 오로지 정보의 제공만을 목적으로 하고, 이로 인한 계약상의 직접적 또는 간접적 책임을 지지 아니하며, 사용설명서 상의 내용은 법적 또는 상업적인 특정한 조건을 만족시키는 것을 보장하지는 않습니다. 사용설명서의 내용은 제품의 업그레이드나 수정에 따라 그 내용이 예고 없이 변경될 수 있으며, 내용상의 오류가 없음을 보장하지 아니합니다.

Trademarks

Tibero® is a registered trademark of TmaxTibero Co., Ltd. Other products, titles or services may be registered trademarks of their respective companies.

Tibero®는 TmaxTibero Co., Ltd.의 등록 상표입니다. 기타 모든 제품들과 회사 이름은 각각 해당 소유주의 상표로서 참조용으로만 사용됩니다.

Open Source Software Notice

Some modules or files of this product are subject to the terms of the following licenses. : OpenSSL, RSA Data Security, Inc., Apache Foundation, Jean-loup Gailly and Mark Adler, Paul Hsieh's hash

Detailed Information related to the license can be found in the following directory : \${INSTALL_PATH}/license/oss_licenses

본 제품의 일부 파일 또는 모듈은 다음의 라이선스를 준수합니다. : OpenSSL, RSA Data Security, Inc., Apache Foundation, Jean-loup Gailly and Mark Adler, Paul Hsieh's hash

관련 상세한 정보는 제품의 다음의 디렉터리에 기재된 사항을 참고해 주십시오. : \${INSTALL_PATH}/license/oss_licenses

안내서 정보

안내서 제목: Tibero tbESQL/C 안내서

발행일: 2025-09-30

소프트웨어 버전: Tibero 7.2.4

안내서 버전: v7.2.4

내용 목차

안내서에 대하여	xv
제1장 tbESQL/C 소개	1
1.1. 개요	1
1.2. 구성요소	1
1.2.1. tbESQL/C 문장	1
1.2.2. 프로그램 변수	2
1.2.3. 구조체 및 배열 변수	3
1.2.4. 커서	4
1.2.5. 프리컴파일러	5
제2장 데이터 타입	7
2.1. 개요	7
2.2. Tiberio 데이터 타입	7
2.3. tbESQL/C 데이터 타입	9
2.3.1. 데이터 타입 대응	9
2.3.2. 데이터 타입 변환	10
2.3.3. 데이터 변수 사용	12
2.3.4. ROWID	14
2.3.5. VARCHAR	15
2.3.6. 구조체	17
2.3.7. 포인터	18
2.3.8. 지시자	20
2.3.9. 데이터 타입 동격화	23
제3장 기본 프로그래밍	25
3.1. 개요	25
3.1.1. tbESQL/C 프로그램의 문법	25
3.1.2. 프로그램의 실행 과정	27
3.1.3. 런타임 에러 처리	29
3.2. 프로그램 구조	29
3.2.1. 변수 선언	30
3.2.2. 초기화	30
3.2.3. 데이터베이스 작업	30
3.2.4. 종료화	31
3.2.5. 에러 처리	31
3.3. tbESQL/C 문장 실행	31
3.3.1. SELECT	32
3.3.2. INSERT	34
3.3.3. UPDATE	35
3.3.4. DELETE	35
3.4. 커서	36

3.4.1.	사용 방법	36
3.4.2.	CURRENT OF 절	38
3.4.3.	사용 예제	39
3.5.	스크롤 가능 커서	41
3.5.1.	사용 방법	41
3.5.2.	사용 예제	42
제4장	배열 변수	45
4.1.	개요	45
4.2.	배열 변수 선언	46
4.3.	입/출력 배열 변수	46
4.3.1.	SELECT	46
4.3.2.	INSERT	53
4.3.3.	UPDATE	55
4.3.4.	DELETE	56
4.3.5.	FOR 절	56
4.4.	구조체 배열 변수	58
4.4.1.	구조체 배열 변수의 선언	58
4.4.2.	사용 방법	59
제5장	멀티 스레드 프로그래밍	61
5.1.	개요	61
5.1.1.	런타임 컨텍스트	61
5.2.	프리컴파일러 옵션과 tbESQL/C 문장	62
5.2.1.	프리컴파일러 옵션	62
5.2.2.	tbESQL/C 문장	62
5.3.	주의사항	63
5.4.	사용 예제	64
제6장	Dynamic SQL	65
6.1.	개요	65
6.2.	특징	65
6.3.	실행 방법	66
6.3.1.	방법 1	66
6.3.2.	방법 2	67
6.3.3.	방법 3	69
6.3.4.	방법 4	72
제7장	런타임 에러 처리	79
7.1.	개요	79
7.2.	상태 변수	79
7.2.1.	특징	79
7.2.2.	상태 변수 선언	80
7.2.3.	상태 변수 종류	81
7.2.4.	사용 예제	82

7.3.	SQLCA	83
7.3.1.	SQLCA 구조체 변수	83
7.3.2.	사용 예제	85
7.4.	WHENEVER	86
7.4.1.	구성요소	86
7.4.2.	사용 예제	88
제8장	tbESQL/C 문장	91
8.1.	개요	91
8.2.	tbESQL/C 문장 문법	91
8.3.	tbESQL/C 문장 공통 문법	92
8.3.1.	AT 절	92
8.3.2.	FOR 절	93
8.3.3.	DESCRIPTOR 이름	93
8.4.	tbESQL/C 문장 목록	95
8.4.1.	ALLOCATE	96
8.4.2.	ALLOCATE DESCRIPTOR	97
8.4.3.	CALL	98
8.4.4.	CLOSE	100
8.4.5.	COMMIT	100
8.4.6.	CONNECT	101
8.4.7.	CONTEXT ALLOCATE	103
8.4.8.	CONTEXT FREE	104
8.4.9.	CONTEXT USE	105
8.4.10.	DEALLOCATE DESCRIPTOR	105
8.4.11.	DECLARE CURSOR	106
8.4.12.	DECLARE DATABASE	107
8.4.13.	DECLARE STATEMENT	108
8.4.14.	DELETE	109
8.4.15.	DESCRIBE	110
8.4.16.	DESCRIBE DESCRIPTOR	111
8.4.17.	ENABLE THREADS	112
8.4.18.	EXECUTE	113
8.4.19.	EXECUTE DESCRIPTOR	114
8.4.20.	EXECUTE ... END-EXEC	116
8.4.21.	EXECUTE IMMEDIATE	117
8.4.22.	FETCH	117
8.4.23.	FETCH DESCRIPTOR	120
8.4.24.	GET DESCRIPTOR	121
8.4.25.	INSERT	124
8.4.26.	LOB CLOSE	125
8.4.27.	LOB CREATE TEMPORARY	125
8.4.28.	LOB DESCRIBE	126

8.4.29.	LOB FREE TEMPORARY	128
8.4.30.	LOB OPEN	129
8.4.31.	LOB READ	130
8.4.32.	LOB WRITE	132
8.4.33.	OPEN	135
8.4.34.	PREPARE	137
8.4.35.	ROLLBACK	138
8.4.36.	SAVEPOINT	139
8.4.37.	SELECT	140
8.4.38.	SET DESCRIPTOR	141
8.4.39.	TYPE	143
8.4.40.	UPDATE	145
8.4.41.	VAR	146
8.4.42.	WHENEVER	148
제9장	tbESQL/C 프리컴파일러 옵션	151
9.1.	개요	151
9.2.	tbESQL/C 프리컴파일러 옵션 지정	151
9.2.1.	환경설정 파일	152
9.2.2.	명령 프롬프트	153
9.2.3.	프로그램 내부	153
9.3.	tbESQL/C 프리컴파일러 옵션 목록	154
9.3.1.	CHAR_MAP	155
9.3.2.	CLOSE_ON_COMMIT	156
9.3.3.	CODE	156
9.3.4.	CONFIG	157
9.3.5.	CPOOL	157
9.3.6.	CMAX	158
9.3.7.	CMIN	158
9.3.8.	CINCR	159
9.3.9.	CTIMEOUT	160
9.3.10.	CNOWAIT	160
9.3.11.	CPP_SUFFIX	161
9.3.12.	DEF_SQLCODE	161
9.3.13.	DEFINE	162
9.3.14.	DYNAMIC	162
9.3.15.	END_OF_FETCH	163
9.3.16.	ERROR_CODE	164
9.3.17.	HOLD_CURSOR	164
9.3.18.	IGNORE_ERROR	165
9.3.19.	INAME	165
9.3.20.	INCLUDE	166
9.3.21.	INSERT_NO_DATA_ERROR	166

9.3.22.	LAZY_PSTMT_CHECK	167
9.3.23.	LINES	168
9.3.24.	LOG_LVL	169
9.3.25.	MODE	170
9.3.26.	ONAME	171
9.3.27.	ORACA	171
9.3.28.	PARSE	172
9.3.29.	PREFETCH	172
9.3.30.	RELEASE_CURSOR	173
9.3.31.	RUNTIME_MODE	173
9.3.32.	SELECT_ERROR	173
9.3.33.	SQLCHECK	174
9.3.34.	SQLERRD_INIT	175
9.3.35.	STMT_CACHE	175
9.3.36.	SYS_INCLUDE	176
9.3.37.	THREADS	176
9.3.38.	TYPE_CODE	177
9.3.39.	UNSAFE_NULL	178
9.3.40.	USERID	178
9.3.41.	VARCHAR_SIZE	179
9.3.42.	WORDSIZE	180
Appendix A.	tbESQL/C 프로그램 예제	181
A.1.	DDL과 PSM	181
A.2.	데이터베이스의 이름	182
A.3.	대용량 객체형	183
색인		187

그림 목차

[그림 1.1]	tbESQL/C 프로그램의 컴파일 과정	5
[그림 2.1]	ROWID의 구성	14
[그림 3.1]	tbESQL/C 프로그램의 실행 과정	27
[그림 8.1]	tbESQL/C 문장의 문법	91

예 목차

[예 1.1]	tbESQL/C에서의 UPDATE 문장	2
[예 1.2]	tbESQL/C에서의 프로그램 변수의 선언	2
[예 1.3]	입/출력 변수의 사용	3
[예 2.1]	데이터 타입의 변환	11
[예 2.2]	TO_CHAR 함수를 이용한 데이터 타입의 변환	12
[예 2.3]	ROWID 타입의 사용	14
[예 2.4]	VARCHAR 타입 변수 선언	15
[예 2.5]	VARCHAR 타입의 잘못된 변수 선언	15
[예 2.6]	VARCHAR 타입이 변환된 구조체	15
[예 2.7]	VARCHAR 변수의 일관성	16
[예 2.8]	포인터 변수의 사용	18
[예 2.9]	출력 변수와 지시자 변수	20
[예 2.10]	입력 변수와 지시자 변수	21
[예 3.1]	DECLARE 영역	25
[예 3.2]	tbESQL/C 프로그램에서의 주석	26
[예 3.3]	tbESQL/C 프로그램에서의 참조	27
[예 3.4]	emp.tbc 프로그램의 프리컴파일	28
[예 3.5]	프리컴파일러 옵션을 사용한 Include 파일의 경로 지정	28
[예 3.6]	컴파일과 링크 과정	28
[예 3.7]	tbsql_error 함수 호출	30
[예 3.8]	입력 변수를 이용해 데이터베이스에 접속	30
[예 3.9]	커서의 선언	36
[예 3.10]	OPEN의 실행	36
[예 3.11]	FETCH의 실행	37
[예 3.12]	WHENEVER 문장의 사용	37
[예 3.13]	CLOSE 사용 예	37
[예 3.14]	커서의 사용	39
[예 4.1]	SELECT 문장에서의 배열 변수의 사용	47
[예 4.2]	sqlca.sqlerrd[2]의 활용	48
[예 4.3]	sqlca.sqlerrd[2]를 활용한 루프의 중단	49
[예 4.4]	INSERT 문장의 배열 변수	54
[예 4.5]	UPDATE 문장의 배열 변수	55
[예 4.6]	DELETE 문장의 배열 변수	56
[예 4.7]	구조체 태그의 삽입	58
[예 7.1]	잘못된 INSERT 문장	88
[예 7.2]	GOTO가 선언된 경우의 무한 루프	89
[예 9.1]	tbESQL/C 프리컴파일 실행	151
[예 9.2]	tbESQL/C 프리컴파일러의 옵션을 사용한 프리컴파일 실행	151

안내서에 대하여

안내서의 대상

본 안내서는 Tibero[®](이하 Tibero)에서 제공하는 tbESQL/C의 기본 개념과 이를 활용한 프로그램의 개발 방법을 알고자 하는 데이터베이스 관리자(Database Administrator, 이하 DBA) 및 애플리케이션 프로그램 개발자를 대상으로 기술한다.

안내서의 전제 조건

본 안내서를 원활히 이해하기 위해서는 다음과 같은 사항을 미리 알고 있어야 한다.

- 데이터베이스의 이해
- RDBMS의 이해
- SQL의 이해
- C 프로그래밍의 이해

안내서의 제한 조건

본 안내서는 Tibero를 실무에 적용하거나 운용하는 데 필요한 모든 사항을 포함하고 있지 않다. 따라서 설치와 환경설정 등 운용 및 관리에 관한 내용은 각 제품 안내서를 참고하기 바란다.

참고

Tibero의 설치 및 환경설정에 관한 내용은 "Tibero 설치 안내서"를 참고한다.

안내서 구성

TiberotbESQL/C 안내서는 총 9개의 장과 Appendix로 이루어져 있다. 각 장의 주요 내용은 다음과 같다.

- 제1장: tbESQL/C 소개

tbESQL/C의 기본 개념과 구성요소를 기술한다.

- 제2장: 데이터 타입

tbESQL/C 프로그램에서 사용되는 데이터 타입과 데이터 타입 간의 대응을 기술한다.

- 제3장: 기본 프로그래밍

tbESQL/C 프로그램의 문법과 실행 과정, 런타임 에러 처리, 그리고 tbESQL/C 문장의 실행과 커서를 기술한다.

- 제4장: 배열 변수

배열 변수의 기본 개념과 선언 방법, tbESQL/C 문장에서 입/출력 변수로 배열 변수를 사용하는 방법을 기술한다.

- 제5장: 멀티 스레드 프로그래밍

tbESQL/C 환경에서 멀티 스레드 프로그램을 작성하는 방법을 기술한다.

- 제6장: Dynamic SQL

tbESQL/C 프로그램이 실행될 때 SQL 문장의 내용을 동적으로 지정하는 방법인 Dynamic SQL을 기술한다.

- 제7장: 런타임 에러 처리

tbESQL/C 프로그램을 실행할 때 발생하는 예외 상황을 처리하기 위한 방법인 런타임 에러 처리를 기술한다.

- 제8장: tbESQL/C 문장

tbESQL/C 프로그램에서 데이터베이스 처리를 위해 사용하는 tbESQL/C 문장을 기술한다.

- 제9장: tbESQL/C 프리컴파일러 옵션

tbESQL/C 프리컴파일러를 동작시킬 때 사용할 수 있는 옵션을 기술한다.

- Appendix A: tbESQL/C 프로그램 예제

DDL과 PSM, 데이터베이스의 이름, 대용량 객체형을 사용하여 tbESQL/C 프로그램을 작성한 예제를 기술한다.

안내서 규약

표기	의미
<<AaBbCc123>>	프로그램 소스 코드의 파일명
<Ctrl>+C	Ctrl과 C를 동시에 누름
[Button]	GUI의 버튼 또는 메뉴 이름
진하게	강조
" "(따옴표)	다른 관련 안내서 또는 안내서 내의 다른 장 및 절 언급
'입력항목'	화면 UI에서 입력 항목에 대한 설명
하이퍼링크	메일 계정, 웹 사이트
>	메뉴의 진행 순서
+----	하위 디렉터리 또는 파일 있음
----	하위 디렉터리 또는 파일 없음
<u>참고</u>	참고 또는 주의사항
<u>주의</u>	주의할 사항
[그림 1.1]	그림 이름
[예 1.1]	예제 이름
AaBbCc123	Java 코드, XML 문서
[command argument]	옵션 파라미터
< xyz >	'<'와 '>' 사이의 내용이 실제 값으로 변경됨
	선택 사항. 예) A B: A나 B 중 하나
...	파라미터 등이 반복되어서 나옴
\${ }	환경변수

시스템 사용 환경

	요구 사항	
Platform	Solaris (Solaris 11)	
	AIX (AIX 7.1/AIX 7.2/AIX 7.3)	
	GNU (X86, 64, IA64)	
	Red Hat Enterprise Linux 7 kernel 3.10.0 이상	
	Windows(x86) 64bit	
	Hardware	최 소 2.5GB 하 드 디 스 크 공 간
4GB 이상 메모리 공간		
Compiler	PSM (C99 지원 필요)	
	tbESQL/C (C99 지원 필요)	

관련 안내서

안내서	설명
Tibero 설치 안내서	설치 시 필요한 시스템 요구사항과 설치 및 제거 방법을 기술한 안내서이다.
Tibero tbCLI 안내서	Call Level Interface인 tbCLI의 개념과 구성요소, 프로그램 구조를 소개하고 tbCLI 프로그램을 작성하는 데 필요한 데이터 타입, 함수, 에러 메시지를 기술한 안내서이다.
Tibero 애플리케이션 개발자 안내서	각종 애플리케이션 라이브러리를 이용하여 애플리케이션 프로그램을 개발하는 방법을 기술한 안내서이다.
Tibero External Procedure 안내서	External Procedure를 소개하고 이를 생성하고 사용하는 방법을 기술한 안내서이다.
Tibero JDBC 개발자 안내서	Tibero에서 제공하는 JDBC 기능을 이용하여 애플리케이션 프로그램을 개발하는 방법을 기술한 안내서이다.
Tibero tbESQL/COBOL 안내서	COBOL 프로그래밍 언어를 사용해 데이터베이스 작업을 수행하는 각종 애플리케이션 프로그램을 작성하는 방법을 기술한 안내서이다.
Tibero tbPSM 안내서	저장 프러시저 모듈인 tbPSM의 개념과 문법, 구성요소를 소개하고, 프로그램을 작성하는 데 필요한 제어 구조, 복합 타입, 서브 프로그램, 패키지과 SQL 문장을 실행하고 에러를 처리하는 방법을 기술한 안내서이다.
Tibero tbPSM 참조 안내서	저장 프러시저 모듈인 tbPSM의 패키지를 소개하고, 이러한 패키지에 포함된 각 프러시저와 함수의 프로토타입, 파라미터, 예제 등을 기술한 참조 안내서이다.
Tibero 관리자 안내서	Tibero의 동작과 주요 기능의 원활한 수행을 보장하기 위해 DBA가 알아야 할 관리 방법을 논리적 또는 물리적 측면에서 설명하고, 관리를 지원하는 각종 도구를 기술한 안내서이다.
Tibero 유틸리티 안내서	데이터베이스와 관련된 작업을 수행하기 위해 필요한 유틸리티의 설치 및 환경설정, 사용 방법을 기술한 안내서이다.
Tibero TAS 안내서	Tibero Active Cluster (TAS)를 사용해서 Tibero의 파일을 관리하고자 하는 관리자를 대상으로 기술한 안내서이다.
Tibero	Tibero를 사용하는 도중에 발생할 수 있는 각종 에러의 원인과 해결 방법을 기술한 안내서이다.

안내서	설명
에러 참조 안내서	
Tibero 참조 안내서	Tibero의 동작과 사용에 필요한 초기화 파라미터와 데이터 사전, 정적 뷰, 동적 뷰를 기술한 참조 안내서이다.
Tibero SQL 참조 안내서	데이터베이스 작업을 수행하거나 애플리케이션 프로그램을 작성할 때 필요한 SQL 문장을 기술한 참조 안내서이다.
Tibero Spatial 참조 안내서	Tibero에서 Geometry 타입에 대한 설명과 Spatial 기능 관련 프러시저 함수 목록 및 사용 방법 등을 기술한 안내서이다.
Tibero TEXT 참조 안내서	Tibero의 제공하는 Text Index를 소개하고, Text Index를 생성 하고 사용하는 방법을 기술하는 안내서이다.
Tibero TDP.NET 안내서	Tibero Data Provider for .NET 기능을 기술하는 안내서이다.
Tibero IMCS 안내서	Tibero에서 제공하는 In-Memory Column Store(이하 IMCS) 기능을 기술하는 안내서이다.

제1장 tbESQL/C 소개

본 장에서는 tbESQL/C의 기본 개념과 tbESQL/C 프로그래밍을 시작하기 전에 알아야 할 구성요소를 설명한다.

1.1. 개요

tbESQL은 ESQL(Embedded SQL: 내장 SQL)의 사용을 위해 Tibero가 제공하는 인터페이스이다. 일반적으로 프로그래밍 언어는 매우 복잡하고 세밀한 작업을 빠르게 수행할 수 있으며, SQL 문장은 간단한 문법만으로 데이터베이스에 직접적인 작업을 수행할 수 있다.

ESQL은 이러한 프로그래밍 언어의 연산 능력과 SQL의 데이터베이스(Database)를 조작하는 능력을 결합하기 위한 방법이며, ANSI 및 ISO 표준으로 정의되어 있다.

Tibero에서는 애플리케이션 개발에 사용되는 C와 COBOL에 대한 tbESQL 인터페이스를 제공한다. C 프로그래밍 언어를 위한 ESQL 인터페이스를 **tbESQL/C**라고 하며, COBOL 프로그래밍 언어에 대한 인터페이스를 tbESQL/COBOL이라고 한다.

참고

tbESQL/COBOL에 대한 내용은 "Tibero tbESQL/COBOL 안내서"를 참고한다.

1.2. 구성요소

1.2.1. tbESQL/C 문장

tbESQL/C 프로그램에는 C 프로그래밍 언어의 소스 코드와 tbESQL/C 문장이 혼합되어 있다. tbESQL/C 프로그램 내에서 SQL 문장의 질의(Query) 등 데이터베이스 처리와 관련된 문장을 tbESQL/C 문장(tbESQL/C Statement)이라고 한다.

tbESQL/C 문장은 일반 SQL 문장과 비슷하지만 다음과 같은 점에서 차이가 난다.

- 항상 EXEC SQL로 시작하고 세미콜론(;)으로 종료한다.
- 필요한 경우에 입력 변수와 출력 변수를 포함한다.
- 일반 SQL 문장에는 없는 새로운 절을 포함할 수 있다. 예를 들어 SELECT 문장은 INTO 절을 포함할 수 있다.

다음은 tbESQL/C 프로그램에서 UPDATE 문장을 작성한 예이다.

[예 1.1] tbESQL/C에서의 UPDATE 문장

```
EXEC SQL UPDATE EMP SET SALARY = SALARY * 1.05
WHERE EMPNO = 5;
```

위의 예에서는 tbESQL/C 문장임을 나타내는 **EXEC SQL**로 시작된다는 점에서 일반 SQL 문장과 차이가 난다는 것을 알 수 있다.

참고

EXEC SQL에 대한 자세한 내용은 [“제8장 tbESQL/C 문장”](#)을 참고한다.

1.2.2. 프로그램 변수

tbESQL/C 프로그램에서 가장 중요한 작업 중에 하나는 프로그램과 데이터베이스 간에 데이터를 전달하는 것이다. 이러한 작업은 주로 프로그램 변수를 이용하여 수행한다. 즉, 프로그램 변수를 이용하여 질의를 하거나 변수의 값을 데이터베이스에 저장할 수 있고, 데이터베이스의 컬럼 값을 프로그램 변수에 저장할 수도 있다. 이러한 작업을 하기 위해서는 Tibero의 데이터 타입과 tbESQL/C 프로그램의 데이터 타입 간의 연관성이 필요하다. 예를 들어 다음의 Tibero의 데이터 타입과 tbESQL/C 프로그램의 데이터 타입은 서로 대응된다.

Tibero의 데이터 타입	tbESQL/C 프로그램의 데이터 타입
NUMBER(p, s)	int, double

NUMBER 타입에서 p는 정밀도(Precision), s는 스케일(Scale)을 의미한다.

참고

Tibero의 데이터 타입과 tbESQL/C 프로그램의 데이터 타입 간의 대응에 대해서는 [“2.3.1. 데이터 타입 대응”](#)을 참고한다.

프로그램 변수 선언

tbESQL/C 프로그램에서의 변수는 C 프로그램의 변수와 거의 동일하게 선언되고 사용된다. 데이터베이스 작업과 관련되지 않는 변수는 C 프로그램에서 사용되는 것과 같이 제약 없이 사용할 수 있다.

다음은 tbESQL/C에서 프로그램 변수를 선언한 예이다.

[예 1.2] tbESQL/C에서의 프로그램 변수의 선언

```
VARCHAR ename[24];
int salary;
VARCHAR addr[32];
```

위의 예에서 VARCHAR 타입은 **tbESQL/C** 프로그램 내에서만 사용할 수 있는 타입이며, 프리컴파일 과정을 거쳐 C 프로그래밍 언어의 데이터 타입으로 변환된다.

입/출력 변수

데이터베이스 작업과 관련된 변수는 다음과 같이 두 가지로 구분된다.

- 입력 변수

tbESQL/C 문장을 통해 컬럼의 값을 삽입, 갱신, 삭제할 때 데이터의 값을 설정하기 위한 변수이다.

입력 변수는 다음과 같은 특징이 있다.

- 입력 변수를 사용할 때는 반드시 변수 앞에 **콜론(:)**을 붙여야 한다.
- 입력 변수는 **SELECT, INSERT, UPDATE, DELETE** 문장과 **WHERE** 절과 **SET** 절 등의 컬럼 값의 위치에 사용될 수 있다.
- 입력 변수는 테이블 이름이나 컬럼 이름의 위치에는 사용될 수 없다.

- 출력 변수

tbESQL/C 문장의 질의 수행 결과로 반환된 값을 저장하기 위한 변수이다.

출력 변수는 다음과 같은 특징이 있다.

- 입력 변수와 마찬가지로 변수 앞에 반드시 **콜론(:)**을 붙여야 한다.
- 출력 변수는 **SELECT** 문장의 **INTO** 절에 사용될 수 있다.
- **INTO** 절에는 출력 변수와 함께 **INDICATOR** 키워드와 지시자 변수가 올 수 있다.

다음은 **WHERE** 절에 사용된 입력 변수 **empno**와 **INTO** 절에 사용된 출력 변수 **ename, salary, addr**를 사용한 예이다.

[예 1.3] 입/출력 변수의 사용

```
EXEC SQL SELECT ENAME, SALARY, ADDR INTO :ename, :salary, :addr
          FROM EMP
          WHERE EMPNO = :empno;
```

위의 문장을 실행하기 전에 **empno** 값을 설정하는 코드가 와야 하며, 또한 문장이 실행된 후에는 반환된 **ENAME, SALARY, ADDR** 컬럼 값에 대한 적절한 작업이 진행되어야 한다.

1.2.3. 구조체 및 배열 변수

구조체와 배열 변수는 동시에 여러 개의 입/출력 변수를 처리하는 자료 구조이다.

구조체

tbESQL/C 프로그램에서는 보통 동시에 여러 개의 입력 변수나 출력 변수가 사용된다. [예 1.3]의 경우에도 SELECT 문장의 INTO 절에서 세 개의 출력 변수(ename, salary, addr)가 사용되었다. 또한 INSERT 문장을 사용할 경우에는 여러 개의 입력 변수가 사용될 수 있다.

이렇게 여러 개의 입력 변수나 출력 변수가 사용될 경우 tbESQL/C 프로그램에서도 C 프로그래밍 언어에 서처럼 여러 개의 변수를 묶어 하나의 구조체로 사용할 수 있다.

참고

입/출력 구조체 변수에 대한 자세한 내용은 “2.3.6. 구조체”를 참고한다.

배열 변수

SELECT 문장을 실행한 결과의 로우 개수는 대개의 경우 하나 이상이며, INSERT 문장을 실행할 때에도 보통 하나 이상의 로우를 삽입하게 된다. 이때 각각의 로우에 대해 개별적으로 SQL 문장을 여러 번 실행하지 않고, 출력 변수나 입력 변수를 배열로 선언하여 SQL 문장을 한 번만 실행할 수도 있다.

출력 변수나 입력 변수를 배열로 선언한 경우 각각을 출력 배열 변수, 입력 배열 변수라고 부른다. 또한 이 두 가지 변수를 한꺼번에 지칭할 때는 입/출력 배열 변수라고 한다. 출력 배열 변수를 이용하여 SELECT 문장을 실행하는 경우 각 결과 로우는 결과 로우의 개수와 동일한 크기를 갖는 출력 배열 변수에 저장된다.

tbESQL/C 프로그램에서는 하나 이상의 결과 로우가 반환되는 SELECT 문장에 출력 배열 변수나 커서를 사용하지 않으면 에러를 반환한다. 구조체도 배열 변수로 선언하여 사용할 수 있으며, 이를 구조체 배열 변수라고 부른다.

참고

입/출력 배열 변수와 구조체 배열 변수에 대한 자세한 내용은 “제4장 배열 변수”를 참고한다.

1.2.4. 커서

커서는 SELECT 문장을 실행한 결과로 반환된 다수의 로우 각각에 차례대로 액세스하는 데이터 구조이다. SELECT 문장을 실행한 결과로 반환되는 로우의 개수는 기본 키에 대한 질의가 아니라면 대부분 하나 이상이다. 또한 반환되는 로우의 개수를 미리 알 수 없는 경우가 많기 때문에 SELECT 문장의 INTO 절에 변수를 하나만 명시해서는 모든 로우의 데이터를 저장할 수 없다. 이러한 경우 프로그램을 보다 간편하고 편리하게 작성하기 위해 앞 절에서 설명한 배열 변수를 사용할 수도 있지만 커서를 사용할 수도 있다.

커서를 사용하는 방법 및 순서는 다음과 같다. 자세한 내용은 “3.4. 커서”를 참고한다.

1. DECLARE CURSOR를 이용하여 커서를 선언한다.
2. OPEN을 실행하여 커서를 연다. 커서를 열면, 연관된 SQL 문장이 실행되고 질의의 결과가 반환된다.

3. FETCH를 실행한다. FETCH를 실행할 때마다 하나 또는 그 이상의 결과 로우를 얻을 수 있다. 커서는 항상 현재 처리 중인 로우를 가리킨다.
4. 모든 결과 로우에 대한 FETCH를 실행한 후에는 CLOSE를 실행하여 커서를 닫는다.

1.2.5. 프리컴파일러

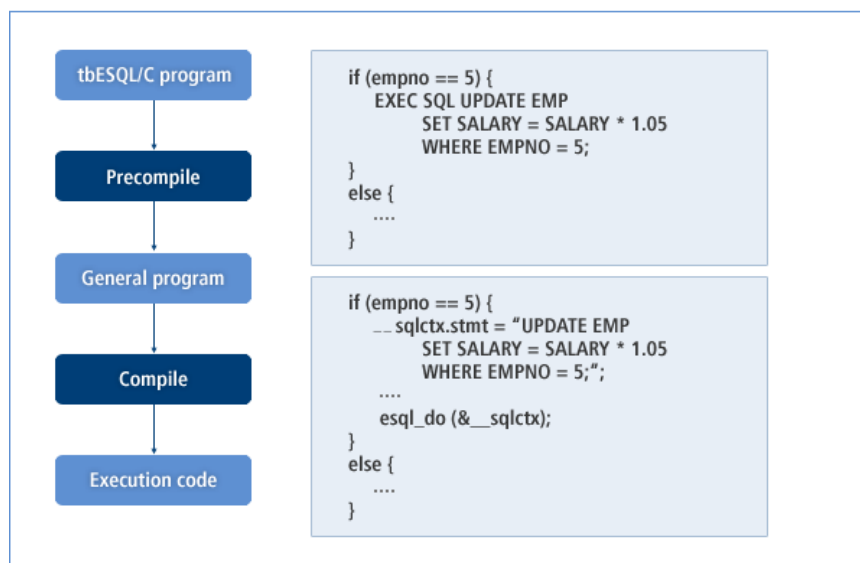
tbESQL/C 프로그램 내에는 C 프로그래밍 언어와 tbESQL/C 문장이 함께 포함되어 있다. tbESQL/C 문장은 C 프로그래밍 언어의 문법을 따르지 않기 때문에 tbESQL/C 프로그램을 컴파일하기 전에 적절한 처리를 해주어야 한다. 이러한 컴파일 이전의 처리 과정을 **프리컴파일(Precompile)**이라고 하며, 그때 사용하는 유틸리티를 **프리컴파일러(Precompiler)**라고 한다.

프리컴파일러는 tbESQL/C 프로그램에 포함된 tbESQL/C 문장을 tbESQL/C 라이브러리 함수로 호출할 수 있는 C 프로그래밍 언어의 소스 코드로 변환해 준다. 이러한 과정을 위해서 tbESQL/C에서는 다음과 같은 함수를 정의하고 있다. 이 함수의 파라미터로는 SQL 문장과 입/출력 변수의 정보가 포함된다.

```
esql_do( )
```

다음은 tbESQL/C 프로그램의 컴파일 과정을 나타내는 그림이다.

[그림 1.1] tbESQL/C 프로그램의 컴파일 과정



tbESQL/C 프로그램의 컴파일 과정은 크게 두 가지 과정으로 나눌 수 있다.

- **프리컴파일(Precompile)**

tbESQL/C 프로그램을 작성하여 프리컴파일 과정을 거치고 나면 C 프로그래밍 언어로만 구성된 소스 코드가 생성된다. 생성된 소스 코드는 **.c** 확장자를 가진 파일의 형태로 저장된다.

- **컴파일(Compile)**

프리컴파일 과정을 거친 프로그램은 다시 컴파일 과정을 거치게 되고 최종적으로 실행 파일이 생성된다.

제2장 데이터 타입

본 장에서는 `tbESQL/C` 프로그램에서 사용하는 데이터 타입을 알아보고 데이터 타입 간의 대응을 설명한다.

2.1. 개요

`tbESQL/COBOL` 프로그램에서 데이터 타입은 `tbESQL/C` 문장에 값을 입력하고, 질의 결과를 얻기 위해 사용한다.

`tbESQL/C`는 다음과 같이 두 가지 타입을 지원한다.

- [Tibero의 데이터 타입](#)

데이터베이스에 저장된 데이터에 접근할 때 사용한다.

- [tbESQL/C의 데이터 타입](#)

애플리케이션 프로그램에서 데이터를 조작할 때 사용한다.

2.2. Tibero 데이터 타입

본 절에서는 Tibero에서 디폴트로 제공하는 데이터 타입을 간략히 설명한다. 이러한 데이터 타입은 데이터베이스의 스키마 객체를 생성하는 데 사용하는 것으로 `tbESQL/C` 프로그램 내에서도 모든 데이터 타입에 대응되는 변수를 사용할 수 있다.

다음은 Tibero의 데이터 타입이다.

구분	데이터 타입	설명
문자형	CHAR, VARCHAR, NCHAR, NVARCHAR, RAW, LONG, LONG RAW	문자열을 표현하는 데이터 타입이다.
숫자형	NUMBER, INTEGER, FLOAT, BINARY_FLOAT, BINARY_DOUBLE	정수나 실수의 숫자를 저장하는 데이터 타입이다.
날짜형	DATE, TIME, TIMESTAMP	시간이나 날짜를 저장하는 데이터 타입이다.
대용량 객체형	BLOB, CLOB	LOB 타입을 의미한다. 다른 데이터 타입이 지원하는 최대 길이(8KB 이하)보다 훨씬 큰 길

구분	데이터 타입	설명
		이를 가질 수 있는 객체이다. 4GB까지 가능하다.
내재형	ROWID	사용자가 명시적으로 선언하지 않아도 Tibero가 자동으로 삽입되는 로우마다 포함하는 컬럼의 타입이다.

참고

자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

다음은 각 데이터 타입에 대한 세부 설명이다.

데이터 타입	설명
CHAR	일반 문자열을 저장하는 데이터 타입이다. (예: CHAR(10))
VARCHAR	일반 문자열을 저장하는 데이터 타입이다. (예: VARCHAR(10))
NCHAR	유니코드 문자열을 저장하기 위한 데이터 타입이다. 타입의 길이는 다국어 문자 집합에 따라 달라진다. (예: NCHAR(10))
NVARCHAR	유니코드 문자열을 저장하기 위한 데이터 타입이다. 타입의 길이는 다국어 문자 집합에 따라 달라진다. (예: NVARCHAR(10))
RAW	임의의 바이너리 데이터를 저장하는 데이터 타입이다. (예: RAW(10))
LONG	VARCHAR 타입을 확장한 데이터 타입이다. VARCHAR 타입과 마찬가지로 문자열이 저장된다. 테이블 내에 한 칼럼에만 선언할 수 있으며, 최대 2GB까지 선언할 수 있다. (예: LONG)
LONG RAW	RAW 타입을 확장한 데이터 타입이다. RAW 타입과 마찬가지로 바이너리 데이터가 저장된다. 테이블 내에 한 칼럼에만 선언할 수 있으며, 최대 2GB까지 선언할 수 있다. (예: LONG RAW)
NUMBER	정수 또는 실수를 저장하는 타입이다. NUMBER 타입을 선언할 때 정밀도와 스케일을 함께 선언할 수 있다. – 정밀도 : 데이터 값의 전체 자릿수 – 스케일 : 소수점 이하 자릿수
INTEGER FLOAT	기본적으로는 NUMBER 타입이다. 단, NUMBER 타입과는 다르게 정밀도와 스케일을 선언할 때 범위에 한계를 둔다. NUMBER 타입의 값은 Tibero에서 가변 길이로 저장되며, 실제 값과 정밀도, 스케일에 따라 그 길이가 달라진다.
BINARY_FLOAT	실수나 정수를 표현하고, 32비트로 저장하는 단일 정밀도 데이터 타입이다.

데이터 타입	설명
	음양으로 절댓값이 1.17549E-38보다 크거나 같고, 3.40282E+38보다 작은 수를 표현할 수 있다.
BINARY_DOUBLE	실수나 정수를 표현하고, 64비트로 저장하는 단일 정밀도 데이터 타입이다. 음양으로 절댓값이 2.22507485850720E-308보다 크거나 같고, 1.79769313486231E+308보다 작은 수를 표현할 수 있다.
DATE	특정 날짜와 시간을 나타내는 데이터 타입이다.
TIME	– DATE : 특정 날짜
TIMESTAMP	– TIME : 특정 시간 – TIMESTAMP : 특정 날짜와 시간
BLOB	임의의 바이너리 데이터를 데이터베이스에 저장하는 데이터 타입이다. 한 테이블의 여러 컬럼에 선언할 수 있다.
CLOB	읽을 수 있는 문자열을 데이터베이스에 저장하는 데이터 타입이다. 한 테이블의 여러 컬럼에 선언할 수 있다.
ROWID	데이터베이스 내의 각 로우를 식별하기 위해 Tibero 시스템이 각 로우마다 자동으로 부여하는 데이터 타입이다. 각 로우가 저장된 물리적인 위치를 포함한다.

2.3. tbESQL/C 데이터 타입

본 절에서는 tbESQL/C의 데이터 타입을 설명한다.

2.3.1. 데이터 타입 대응

Tibero에서 제공하는 데이터 타입을 tbESQL/C 프로그램에서 그대로 사용할 수는 없다. tbESQL/C에는 Tibero의 각 데이터 타입에 대응되는 tbESQL/C의 데이터 타입이 정의되어 있다.

tbESQL/C의 데이터 타입은 대체로 C 프로그래밍 언어의 데이터 타입과 동일하다. 또한 각 Tibero의 데이터 타입에 대응되는 tbESQL/C의 데이터 타입은 하나 이상일 수도 있다.

다음은 Tibero의 데이터 타입에 대응되는 tbESQL/C의 데이터 타입이다.

Tibero의 데이터 타입	tbESQL/C의 데이터 타입	설명
CHAR, VARCHAR	char	하나의 문자
	char[n]	길이 n의 문자열

Tibero의 데이터 타입	tbESQL/C의 데이터 타입	설명
RAW	varchar[n]	길이 n의 문자열
	unsigned char[n]	길이 n의 바이너리 데이터
NUMBER	varchar[n]	길이 n의 바이너리 데이터
	int, short, long	정수 데이터
DATE, TIME, TIMESTAMP	float, double	실수 데이터 (데이터가 삽입될 때 이미 정해진 컬럼의 정밀도 및 스케일을 초과할 수 있다.)
	char[n]	길이 n의 문자열로 변환
	varchar[n]	길이 n의 문자열로 변환
	unsigned char[n]	길이 n의 바이너리 데이터
ROWID	varchar[n]	길이 n의 문자열 또는 바이너리 데이터
	unsigned char[n]	길이 n의 문자열로 변환
	varchar[n]	길이 n의 문자열로 변환

tbESQL/C의 데이터 타입 중 **VARCHAR 타입**은 Tibero의 데이터 타입 중에 VARCHAR 타입을 비롯한 여러 가지 타입에 대응하기 위해 새롭게 정의된 타입이다.

tbESQL/C 프로그램에서는 각 데이터 타입 간의 변환을 지원한다. 예를 들어 VARCHAR 타입의 문자열이 정수를 표현하고 있는 내용이라면, 그 값을 int 타입의 변수에 저장할 수 있다.

또한 실제로 DATE, TIME, TIMESTAMP 타입과 ROWID 타입에 바로 대응되는 타입은 없으며, 항상 변환 과정을 거쳐서 저장해야 한다.

참고

VARCHAR 타입의 자세한 내용은 [“2.3.5. VARCHAR”](#)를 참고한다.

2.3.2. 데이터 타입 변환

tbESQL/C 프로그램에서는 Tibero 데이터 타입 각각에 대응되는 타입 이외에 다른 데이터 타입을 사용할 수 있다. 예를 들면 데이터베이스의 NUMBER 타입의 컬럼 값을 저장하기 위해 tbESQL/C 프로그램에서는 출력 변수로 VARCHAR 타입을 사용할 수 있다. 이와 반대로, NUMBER 타입 프로그램 변수의 값을 VARCHAR 타입의 컬럼에 저장할 수도 있다.

tbESQL/C 프로그램을 프리컴파일하면 입/출력 변수의 데이터 타입이 프로그램 내에 함께 포함된다. 이렇게 포함된 데이터 타입을 기준으로 데이터의 값이 입/출력될 때 컬럼 타입과 비교하여 필요한 경우에는 데이터 타입의 변환을 수행한다. 만약 데이터 타입의 변환이 불가능한 경우에는 에러를 반환한다.

변환 가능한 데이터 타입

다음은 Tiberio 데이터 타입으로부터 변환 가능한 **tbESQL/C** 데이터 타입이다. 자세한 내용은 “2.3.1. 데이터 타입 대응”에 일부 내용이 포함되어 있다.

Tiberio의 데이터 타입	tbESQL/C의 데이터 타입	설명
NUMBER	int, short, long	정수 데이터
	float, double	실수 데이터
CHAR, VARCHAR	char	문자 데이터
	char[n], varchar[n]	문자열 데이터
DATE, TIME, TIMESTAMP	char[n], varchar[n]	날짜형 데이터
ROWID	unsigned char[n], varchar[n]	ROWID 데이터

tbESQL/C 데이터 타입으로부터 Tiberio 데이터 타입으로 변환할 때에도 위의 변환 관계가 적용된다. 예를 들어 tbESQL/C 프로그램의 int 타입의 변수 값을 VARCHAR 타입 컬럼에 저장할 수 있으며, VARCHAR 타입 값을 이용하여 ROWID에 대한 질의를 수행할 수 있다.

다음은 데이터 타입의 변환을 수행하는 예이다.

[예 2.1] 데이터 타입의 변환

```
VARCHAR sal_str[8];
VARCHAR emp_date[20];
...
EXEC SQL SELECT SALARY, EMP_DATE
      INTO :sal_str, :emp_date
      FROM EMP
      WHERE EMPNO = 20;

printf("salary = %.s\n", sal_str.len, sal_str.arr);
printf("emp_date = %.s\n", emp_date.len, emp_date.arr);
```

위의 [예 2.1]을 실행하면 다음과 같은 내용이 출력된다.

```
salary = 35000
emp_date = 2001-12-01
```

데이터 타입을 변환할 때에는 데이터 값의 범위와 내용에 유의해야 한다. 데이터 값의 범위는 그 값을 저장할 장소가 충분히 포함할 수 있는 한도 내이어야 한다.

데이터 값의 범위와 관련된 예를 들면 tbESQL/C 프로그램의 short 타입의 변수에 short 타입의 범위를 넘어서는 값인 문자열 "327680"을 변환하거나, 마찬가지로 VARCHAR(3)의 타입을 갖는 컬럼에 그 범위를 넘어서는 값인 65535를 변환할 수 없다.

데이터의 내용에 관한 예를 들면 데이터베이스 컬럼에 저장된 "ABCDE" 문자열을 프로그램 short 타입의 변수에 변환할 수 없으며, 프로그램 변수에 저장된 "가나다" 문자열을 DATE 타입의 컬럼에 변환할 수 없다.

내장 함수를 이용한 데이터 타입의 변환

Tibero **내장 함수**를 이용하여 데이터 타입의 변환을 실행할 수도 있다. 그러한 함수로는 **TO_CHAR**, **TO_DATE**, **TO_NUMBER** 등이 있다.

다음은 TO_CHAR 함수를 이용하여 실수 데이터를 문자 데이터로 변환하여 출력하는 예이다.

[예 2.2] TO_CHAR 함수를 이용한 데이터 타입의 변환

```
VARCHAR sal_str[16];
...
EXEC SQL SELECT TO_CHAR(SALARY, '$99,999.99') ... ① ...
           INTO :sal_str ... ② ...
           FROM EMP
           WHERE EMPNO = 20;

printf("salary = %.s\n", sal_str.len, sal_str.arr); ... ③ ...
```

① 실수 데이터를 담고 있는 변수 SALARY를 TO_CHAR 함수를 통해 형식 문자열('\$99,999.99')을 지정하여 문자 데이터로 변환한다.

② VARCHAR 타입 변수 sal_str에 저장한다.

③ printf 문을 통해 sal_str 변수를 출력한다.

위의 [예 2.2]을 실행하면 다음과 같은 내용이 출력된다.

```
salary = $35,000.00
```

참고

데이터 타입을 변환하는 내장 함수에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

2.3.3. 데이터 변수 사용

C 프로그래밍 언어의 변수와는 달리 **tbESQL/C** 프로그램에서 데이터베이스 작업과 관련된 변수는 모두 **DECLARE 영역** 내에 선언되어야 한다.

다음은 DECLARE 영역 내에 선언된 변수의 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR ename[24];
```

```
int salary;
VARCHAR addr[32];
int empno;
EXEC SQL END DECLARE SECTION;
```

위의 예에서 알 수 있듯이 DECLARE 영역은 'EXEC SQL BEGIN DECLARE SECTION;'으로 시작하고 'EXEC SQL END DECLARE SECTION;'으로 끝난다.

VARCHAR 타입은 일반적인 CHAR 배열 타입과 유사하게 선언한다. CHAR 타입과는 달리 배열이 아닌 형태로는 선언이 불가능하다. 프리컴파일 과정을 거치면 VARCHAR 타입은 tbESQL/C에서 정의한 구조체 타입으로 변환된다.

DECLARE 영역 내에 선언된 변수는 C 프로그래밍 언어의 변수와 동일한 방법으로 프로그램 내에서 사용된다. 하지만 tbESQL/C 문장 내에서의 변수는 tbESQL/C 문장과 구별을 위하여 반드시 콜론(:) 뒤에 와야 한다. 이렇게 콜론(:) 뒤에 사용된 tbESQL/C 문장 내의 변수를 **입/출력 변수**라고 한다.

다음은 tbESQL/C 문장 내에서 사용된 입력 변수와 출력 변수에 대한 예이다.

```
empno = 20;
EXEC SQL SELECT ENAME, SALARY, ADDR
      INTO :ename, :salary, :addr
      FROM EMP
      WHERE EMPNO = :empno;

printf("salary = %d\n", salary);
```

위의 예에서는 SELECT 문장을 실행하기 위해 먼저 입력 변수 empno의 값을 읽어와 tbESQL/C 문장을 완성한다. 그리고 나서, SELECT 문장을 실행하고 실행 결과로 반환된 로우의 각 컬럼 값이 출력 변수 ename, salary, addr에 할당된다. 출력 변수 ename, salary, addr은 C 프로그래밍 언어의 변수와 마찬가지로 사용될 수 있다.

만약 SELECT 문장의 실행 결과로 반환된 로우가 없거나 둘 이상의 로우가 반환되면 에러가 발생한다. 이러한 경우 에러를 처리하는 루틴이 미리 정의되어 있으면 그 루틴을 실행하게 되고, 그렇지 않으면 프로그램을 종료한다.

tbESQL/C 프로그램의 SQL 문장 내에는 변수 이외에 상수 또는 함수를 사용할 수는 없다. 따라서 다음은 잘못된 소스 코드이다.

```
#define MAX_SALARY 50000
...
char *get_ename(...);
...
EXEC SQL INSERT INTO EMP(ENAME, SALARY)
      VALUES (:get_ename(), :MAX_SALARY);
```

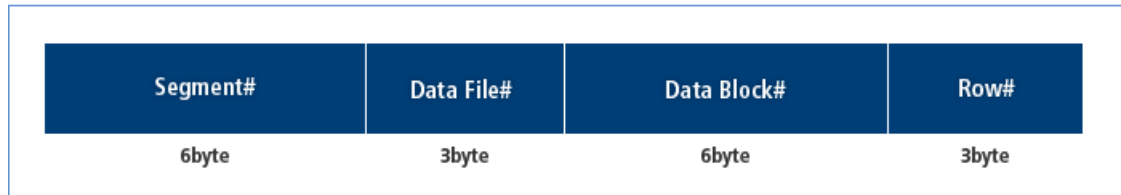
위의 소스 코드는 사용자 정의 함수 get_ename과 상수 MAX_SALARY를 사용했으므로 잘못된 예이다.

2.3.4. ROWID

ROWID 타입은 로우의 물리적인 위치 정보를 포함하는 데이터 타입이다. **tbESQL/C**에서는 **ROWID**를 위한 별도의 데이터 타입을 제공하지 않는다. **unsigned char** 타입 또는 **VARCHAR** 타입을 이용해 데이터 타입을 변환하여 사용해야 한다.

ROWID 타입의 값은 다음의 그림과 같이 4부분으로 구성된다.

[그림 2.1] ROWID의 구성



문자열 변수에 저장되는 ROWID 값은 전체 18bytes를 가지므로, 문자열 변수의 길이는 NULL 값을 포함하여 최소한 19bytes가 되어야 한다. Tibero의 데이터베이스에서는 다른 형태로 저장된다.

참고

ROWID 타입에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

다음은 ROWID 타입을 사용하는 예이다.

[예 2.3] ROWID 타입의 사용

```
EXEC SQL BEGIN DECLARE SECTION;
unsigned char rowid[20];
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT ROWID, ...
      INTO :rowid, ...
      FROM EMP
      WHERE EMPNO = :empno;

printf("rowid = %s\n", rowid);
```

ROWID의 값은 이처럼 출력 변수로 사용될 뿐만 아니라 **SELECT** 문장의 **WHERE** 절이나 **INSERT** 문장에서 입력 변수로 사용될 수도 있다.

위의 [예 2.3]을 실행하면 다음과 같은 내용이 출력된다.

```
rowid = AAAABkAAUAAAAD6AAA
```

2.3.5. VARCHAR

VARCHAR 타입은 Tiberio의 데이터 타입의 VARCHAR 타입을 `tbESQL/C` 프로그램에서 사용하기 위해 새롭게 정의한 데이터 타입이다. VARCHAR 타입은 RAW, DATE, ROWID 타입에도 대응하여 사용할 수 있다.

VARCHAR 타입 변수 선언

VARCHAR 타입 변수 선언은 C 프로그래밍 언어에서 `char` 타입의 배열 변수를 선언하는 것과 동일하다. 한번에 하나의 변수만 선언할 수도 있고, 여러 변수를 함께 선언할 수도 있다. 이때 배열의 크기를 반드시 지정해 주어야 하며, CHAR 타입처럼 문자 하나만 저장하는 용도로는 사용할 수 없다.

다음은 VARCHAR 타입 변수를 선언하는 예이다.

[예 2.4] VARCHAR 타입 변수 선언

```
VARCHAR username[16];
VARCHAR ename[20], addr[24];
```

다음은 VARCHAR 타입 변수가 잘못 선언된 경우의 예이다.

[예 2.5] VARCHAR 타입의 잘못된 변수 선언

```
VARCHAR ch, dname[];
```

변수 **ch**의 경우 VARCHAR 타입은 CHAR 타입처럼 문자 하나만 저장하는 용도로 사용할 수 없는데, 마치 CHAR 타입의 변수를 선언하는 것처럼 선언했기 때문에 잘못되었다. 변수 **dname**의 경우 VARCHAR 타입 변수를 선언할 때에는 반드시 배열의 크기를 정해주어야 하는데, 크기를 정하는 부분을 공란으로 두었기 때문에 잘못되었다.

위의 [예 2.4]에서 선언된 VARCHAR 타입 변수는 프리컴파일러를 통하여 다음과 같은 구조체 타입의 변수로 변환된다.

[예 2.6] VARCHAR 타입이 변환된 구조체

```
struct
{
    unsigned short len;
    unsigned char arr[16];
} username;
```

VARCHAR 타입 변수의 참조와 일관성

VARCHAR 타입 변수를 선언할 때를 제외하고, VARCHAR 타입 변수를 사용하기 위해서는 프리컴파일러가 변환한 구조체의 문법을 따라야 한다.

예를 들어 위의 [예 2.4]에서 선언한 `username`을 출력하고자 한다면, 프리컴파일러에 의해 변환된 [예 2.6]의 구조체를 출력하는 문법에 맞춰 다음과 같이 코드를 작성해야 한다.

```
printf("%.s\n", username.len, username.arr);
```

혹은 아래와 같이 작성할 수 있다.

```
username.arr[username.len] = '\0';  
printf("%s", username.arr);
```

위의 예에서 `username`의 내용을 출력하기 위해서 구조체의 멤버 변수를 참조하는 방법과 마찬가지로 점(.)을 사용한 것을 알 수 있다.

VARCHAR 구조체 내의 변수 `len`과 `arr`은 입력 변수일 때 항상 일관성을 유지해야 한다. 즉, 다음과 같은 조건을 만족해야 한다. 출력 변수일 경우 아래 조건을 만족하지 않는다.

[예 2.7] VARCHAR 변수의 일관성

```
strlen(username.arr) == username.len
```

VARCHAR 타입 변수	설명
입력 변수	입력 변수로 사용된 경우 <code>tbESQL/C</code> 프로그램 내에 조건을 유지시키는 코드를 작성해야 한다. 조건을 만족하지 않는다면 <code>tbESQL/C</code> 라이브러리에서는 <code>len</code> 변수를 우선적으로 참조한다.
출력 변수	출력 변수로 사용된 경우 <code>tbESQL/C</code> 라이브러리는 자동으로 <code>len</code> 멤버를 설정하지만 <code>arr</code> 멤버는 널 종료하지 않는다.

NULL 값의 처리

VARCHAR 타입 변수의 값이 NULL인 경우에는 프리컴파일러를 통해 변환된 구조체 내의 변수 `arr`과 `len`의 값은 다음과 같다.

```
arr == ""  
len == 0
```

VARCHAR 타입 변수의 값이 NULL일 때 VARCHAR 타입 변수가 입력 변수로 사용되었는지, 출력 변수로 사용되었는지에 따라 변환된 구조체 내의 멤버 변수 `arr`과 `len`의 값은 다르다.

VARCHAR 타입 변수	설명
입력 변수	입력 변수인 경우에는 len 변수에 0을 할당하는 코드를 작성해야 한다. arr 문자열의 길이가 0이 아니더라도 len 변수를 먼저 참조하므로 NULL로 인식한다.
출력 변수	출력 변수인 경우에는 arr과 len의 값을 tbESQL/C 라이브러리에서 자동으로 설정해준다.

다음은 VARCHAR 타입의 출력 변수 addr의 값으로 NULL이 반환되었는지 검토하여 출력하는 예이다.

```

VARCHAR addr[32];

...

EXEC SQL SELECT ADDR INTO :addr
      FROM EMP
      WHERE EMPNO = 20;

if (arr.len > 0) {
    addr.arr[addr.len] = '\0';
}

printf("addr = %s\n", (addr.len > 0) ? (addr.arr) : ("(NULL)"));

```

2.3.6. 구조체

tbESQL/C 프로그램에서도 C 프로그래밍 언어의 **구조체**를 사용할 수 있다.

tbESQL/C 프로그램의 SELECT 문장에서는 질의 결과로 반환되는 컬럼의 개수만큼 INTO 절에 출력 변수를 명시해야 한다. 이런 경우에 INTO 절에 명시될 다수의 출력 변수를 한데 묶어 구조체를 만들고, INTO 절에 이 구조체 변수 하나만 명시해 프로그램을 간소화할 수 있다.

구조체 변수를 사용할 때에 SELECT 문장의 결과 로우 내의 컬럼의 순서와 구조체 변수 내의 변수의 순서가 같아야 한다는 것에 유의한다.

다음은 세 개의 출력 변수를 포함하는 구조체를 사용한 예이다.

```

EXEC SQL BEGIN DECLARE SECTION;

struct
{
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[32];
}

```

```

} emp;

...

EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT ENAME, SALARY, ADDR
        INTO :emp
        FROM EMP
        WHERE EMPNO = :empno;

```

구조체 변수는 INSERT 문장 등에서 입력 변수로 사용될 수도 있다. 이때에도 출력 변수와 동일하게 하나의 구조체 변수만 사용할 수 있으며, 구조체 내부에 삽입하려는 컬럼과 같은 순서로 변수가 정의되어 있어야 한다.

다음은 세 개의 컬럼에 값을 삽입하는 예이다.

```

strcpy(emp.ename.arr, "Smith");
emp.salary = 35000;
strcpy(emp.addr.arr, "Los Angeles");
emp.addr.len = strlen(emp.addr.arr);

EXEC SQL INSERT INTO EMP(ENAME, SALARY, ADDR) VALUES (:emp);

```

2.3.7. 포인터

tbESQL/C 프로그램 내의 입/출력 변수에 대하여 C 프로그래밍 언어의 **포인터**를 사용할 수 있다. 예를 들어 다음의 프로그램 코드와 같이 SELECT 문장의 INTO 절에 일반 변수가 아닌 포인터 변수를 사용할 수 있다.

[예 2.8] 포인터 변수의 사용

```

EXEC SQL BEGIN DECLARE SECTION;
int salary, *ptr_salary = NULL;    ... ① ...
...
EXEC SQL END DECLARE SECTION;

...

ptr_salary = &salary;              ... ② ...

EXEC SQL SELECT SALARY, ...

```

```

    INTO :ptr_salary, ...          ... ③ ...
    FROM EMP WHERE EMPNO = 5;

```

- ① 포인터 변수 ptr_salary를 선언한다.
- ② 변수 salary의 주소를 ptr_salary에 할당한다.
- ③ 포인터 변수 ptr_salary를 SELECT 문장의 INTO 절에 출력 변수로 사용한다.

위의 [예 2.8]과 같은 tbESQL/C 프로그램이 프리컴파일 과정을 거치면 SQL 문장 내의 입/출력 변수는 모두 포인터 타입의 변수로 변환된다. tbESQL/C 라이브러리는 포인터 변수의 데이터 타입에 맞춰 입/출력 데이터 값을 적절하게 할당해 준다.

tbESQL/C 프로그램에는 C 프로그래밍 언어에서처럼 일반 변수를 포인터 변수로 변환하기 위한 연산자(&)를 넣어서는 안 된다. 따라서 다음은 잘못된 프로그램 코드이다.

```

EXEC SQL SELECT SALARY, ...
        INTO :&salary, ...
        FROM EMP WHERE EMPNO = 5;

```

구조체 변수에 대한 포인터 변수도 사용할 수 있다.

다음은 구조체 포인터 변수를 사용하는 예이다.

```

EXEC SQL BEGIN DECLARE SECTION;

struct
{
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[32];
} emp, *ptr_emp = NULL;          ... ① ...

...

EXEC SQL END DECLARE SECTION;

...

ptr_emp = &emp;                  ... ② ...

EXEC SQL SELECT ENAME, SALARY, ADDR
        INTO :ptr_emp            ... ③ ...
        FROM EMP
        WHERE EMPNO = :empno;

```

- ① 구조체 타입의 변수 emp와 구조체 타입의 포인터 변수 ptr_emp를 선언한다.

② 구조체 변수 emp의 주소를 포인터 변수 ptr_emp에 저장한다.

③ 포인터 변수 ptr_emp를 SELECT 문장의 INTO 절에서 출력 변수로 사용한다. 이때 & 연산자를 사용하지 않고, 일반 입/출력 변수를 사용할 때와 동일하게 콜론(:)만을 사용한다는 점에 주의해야 한다.

2.3.8. 지시자

일반 프로그램과 달리 tbESQL/C 프로그램에서만 사용되는 변수로 **지시자(INDICATOR)** 변수가 있다. tbESQL/C 문장을 통해 데이터베이스와 tbESQL/C 프로그램 간에 데이터를 주고받을 때 지시자 변수는 전달된 데이터에 대한 정보를 저장하고 있다.

지시자 변수의 선언

지시자 변수는 short 타입을 가지며, 반드시 **DECLARE 영역** 안에 선언되어야 한다.

SELECT 문장에 사용된 지시자 변수는 INTO 절에서 INDICATOR 키워드와 콜론(:) 다음에 오거나, INDICATOR 없이 데이터 변수 바로 뒤에 콜론(:)과 함께 올 수도 있다.

다음은 SELECT 문장에서 출력 변수와 지시자 변수가 사용된 예이다.

[예 2.9] 출력 변수와 지시자 변수

```
EXEC SQL BEGIN DECLARE SECTION;
short ename_ind, addr_ind;
...
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT ENAME, ADDR
      INTO :ename INDICATOR :ename_ind, :addr INDICATOR :addr_ind
      FROM EMP
      WHERE EMPNO = :empno;
```

위의 예에서는 출력 변수 ename에 대응되는 지시자 변수로 ename_ind가 사용되었고, 출력 변수 addr에 대응되는 지시자 변수로는 addr_ind가 사용되었다.

INDICATOR 키워드를 명시하지 않고 다음과 같이 작성해도 위의 [예 2.9]의 문장과 동일한 의미를 갖는다.

```
EXEC SQL SELECT ENAME, ADDR
      INTO :ename:ename_ind, :addr:addr_ind
      FROM EMP
      WHERE EMPNO = :empno;
```

다음은 출력 변수와 함께 사용된 지시자 변수 값의 의미를 정리한 표이다. **tbESQL/C** 프로그램에서는 필요한 경우 지시자 변수의 값을 검토하여 그 값에 따른 처리를 해야 한다.

지시자 변수의 값	설명
0	데이터 값이 성공적으로 저장되었다.
-1	데이터 값이 NULL이다.
-2	데이터의 길이를 미리 알 수가 없다. 길이를 미리 알 수 없는 경우는 2가지가 있다. 첫번째 경우는 BLOB, CLOB 같이 매우 큰 데이터를 저장할 수 있는 데이터 타입을 사용하거나 LONG 타입을 사용한 경우이다. 두번째 경우는 column 에 들어있는 데이터 크기가 32767를 넘을 경우이다.
> 0	문자열 데이터 변수에 저장된 값이 잘린(truncated) 값이다. 지시자 변수에 주어진 값은 실제 데이터베이스에 저장된 문자열의 길이이다.

다음은 **INSERT** 문장에서 입력 변수와 함께 지시자 변수를 사용한 예이다.

[예 2.10] 입력 변수와 지시자 변수

```

VARCHAR ename[24];
VARCHAR addr[36];

short addr_ind;
int empno;

...

strcpy(ename.arr, "Jaochim");
ename.len = strlen(ename.arr);
strcpy(addr.arr, "New York");
addr.len = strlen(addr.arr);

addr_ind = -1;
empno = 25;

EXEC SQL INSERT INTO EMP (ENAME, ADDR, EMPNO)
VALUES (:ename, :addr INDICATOR :addr_ind, :empno);

```

위의 예에서는 입력 변수로 **ename**, **addr**, **empno**가 사용되었으며, 입력 변수 **addr**에 대응되는 지시자 변수로 **addr_ind**가 사용되었다.

INDICATOR 키워드를 명시하지 않고 다음과 같이 작성해도 위의 [예 2.10]의 문장과 동일한 의미를 갖는다.

```
EXEC SQL INSERT INTO EMP (ENAME, ADDR, EMPNO)
VALUES (:ename, :addr:addr_ind, :empno);
```

위의 예에서는 INDICATOR 키워드를 생략하고 입력 변수 `addr` 뒤에 지시자 변수 `addr_ind`를 바로 붙여서 명시하였다.

지시자 변수 값이 -1인 경우에는 입력 변수의 값이 NULL이라는 의미이다. 이때 입력 변수에 저장된 실제 값은 무시된다. 따라서 지시자 변수의 값이 -1인 경우 앞의 INSERT 문장은 다음과 같이 고쳐 써도 된다.

```
EXEC SQL INSERT INTO EMP (ENAME, ADDR, EMPNO)
VALUES (:ename, NULL, :empno);
```

지시자 변수 값이 -1인 경우는 입력 변수 `addr`의 값이 NULL이라는 것이므로 입력 변수와 지시자 변수를 명시할 필요 없이 NULL만 명시해도 된다.

다음은 입력 변수와 함께 사용된 지시자 변수 값의 의미를 정리한 표이다.

지시자 변수의 값	설명
-1	데이터 값이 NULL이다.
>= 0	입력 변수에 저장된 값을 그대로 사용한다.

지시자 변수를 사용하지 않고 `tbESQL/C` 프로그램을 작성할 수도 있지만, SQL 문장의 질의 결과로 반환 되는 값에 대해 충분히 알고 있지 않다면 지시자 변수를 사용하여 검토하는 코드를 삽입하는 것이 좋다.

구조체 타입의 지시자

SELECT 문의 INTO 절에 구조체 변수와 지시자 변수를 함께 사용하는 경우 지시자 변수 역시 마찬가지로 별도의 구조체 변수로 구성해야 한다. 이러한 지시자 변수를 **구조체 타입의 지시자(STRUCTURAL INDICATOR)**라고 한다. 구조체 타입의 지시자도 출력 구조체 변수를 구성하는 것과 마찬가지로 질의 결과 컬럼과 같은 순서로 지시자 변수가 와야 한다. 또한 모든 지시자 변수는 short 타입을 갖는다.

다음은 구조체 타입의 지시자 변수를 사용하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;

struct
{
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[32];
} emp;

struct
{
    short ename_ind;
```

```

    short sal_ind;                ... ② ...
    short addr_ind;
} emp_ind;

...

EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT ENAME, SALARY, ADDR
      INTO :emp INDICATOR :emp_ind
      FROM EMP
      WHERE EMPNO = :empno;

```

- ① 컬럼 ENAME, SALARY, ADDR의 내용을 저장하기 위해서 구조체 타입의 변수로 **emp**를 선언한다.
- ② 구조체 변수를 **SELECT** 문장의 출력 변수로 사용하면서, 이에 대응되는 지시자 변수 **emp_ind** 역시 구조체 변수로 선언한다. 구조체 변수 **emp_ind**를 정의할 때 구조체 내의 멤버 변수를 **emp**에 대응되게 정의하고, **emp_ind**의 모든 멤버 변수를 **short** 타입으로 정의한다.

2.3.9. 데이터 타입 동격화

데이터 타입의 동격화(Datatype Equivalencing)는 Tiberio가 입력 데이터의 해석 방법을 프로그래머가 원하는 대로 변경시키는 것이다. 선언된 하나 하나의 호스트 변수마다 동격화를 시킬 수 있고, **tbESQL/C**에서는 사용자 정의 데이터 타입을 외부 데이터 타입으로 한 번에 변경시킬 수 있다.

다음은 데이터 타입 동격화를 위한 **EXEC SQL TYPE** 구문을 사용한 예이다.

```

EXEC SQL TYPE 사용자정의타입
      IS 외부데이터타입 [(길이)] [REFERENCE];

typedef char asciz[350];
EXEC SQL BEGIN DECLARE SECTION;
      EXEC SQL TYPE asciz IS STRING(350) REFERENCE;
      asciz  catalog_cd[100];    /* ARRAY 변수 정의 */
      asciz  factor_cd1;
EXEC SQL END DECLARE SECTION;

```

다음은 데이터 타입의 동격화 중 사용 가능한 외부 데이터 타입들이다.

```

EXEC SQL TYPE var1 IS
      (STRING | VARRAW | LONG VARRAW | VARCHAR | LONG VARCHAR | RAW) REFERENCE;

```

다음은 데이터 타입 동격화를 위한 **EXEC SQL VAR** 구문을 사용한 예이다.

```
EXEC SQL VAR 사용자변수
      IS 외부데이터타입 [(길이)];

char raw_var[1000];
EXEC SQL VAR raw_var IS RAW(1000);
```

다음은 데이터 타입의 동격화 중 사용 가능한 외부 데이터 타입들이다.

```
EXEC SQL VAR var1 IS
 (STRING | VARRAW | LONG VARRAW | VARCHAR | LONG VARCHAR | RAW);
```

제3장 기본 프로그래밍

본 장에서는 tbESQL/C 프로그램의 문법과 실행 과정, 런타임 에러(runtime error) 처리, 그리고 tbESQL/C 문장의 실행, 커서를 설명한다.

3.1. 개요

본 절에서는 tbESQL/C 프로그램의 문법과 런타임 에러 처리에 대해서 설명한다.

3.1.1. tbESQL/C 프로그램의 문법

tbESQL/C 프로그램의 문법은 다음과 같다.

- SQL 문장의 시작과 끝

- tbESQL/C 프로그램에 포함되는 SQL 문장은 항상 **EXEC SQL**로 시작되며 **세미콜론(;)으로** 끝난다.
- 하나의 SQL 문장은 여러 줄에 걸쳐있을 수 있다.

- DECLARE 영역

- DECLARE 영역은 **'BEGIN DECLARE SECTION;'**으로 시작되며, **'END DECLARE SECTION;'**으로 끝난다.
- DECLARE 영역에는 변수 선언 이외에 다른 코드가 삽입되어서는 안 된다.
- SQL 문장과 함께 사용되는 입/출력 변수는 항상 DECLARE 영역에 선언해야 한다.

입/출력 변수가 구조체나 배열의 형태로 선언된 경우에도 DECLARE 영역에 선언해야 한다. 단, 프리컴파일러 옵션에 따라 그렇지 않은 경우도 있다. 프리컴파일러 옵션에 대해서는 [“제9장 tbESQL/C 프리컴파일러 옵션”](#)을 참고한다.

- 입/출력 변수가 아닌 일반적인 프로그램 변수의 경우에는 DECLARE 영역 밖에 선언되어도 무방하다.
- 다음은 DECLARE 영역의 예이다.

[예 3.1] DECLARE 영역

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[32];
EXEC SQL END DECLARE SECTION;
```

- 문자열

- C 프로그래밍 코드에 포함된 문자열은 큰따옴표(" ")를 사용한다.
- `tbESQL/C` 문장에 포함되는 문자열은 작은따옴표(' ')를 사용한다.

- 연산자

- SQL 문장 내에서는 SQL 문장의 표준으로 정의된 연산자만 사용할 수 있으며, C 프로그래밍 언어의 연산자는 사용할 수 없다. 예를 들어 비트 연산자(~, &, |, ^ 등)는 SQL 문장 내에서 사용될 수 없다.

다음은 SQL 논리 연산자와 C 프로그래밍 언어의 논리 연산자를 비교한 것이다.

SQL 연산자	C 프로그래밍 언어의 연산자	설명
NOT	!	TRUE이면 FALSE, FALSE이면 TRUE를 반환한다.
AND	&&	둘 다 TRUE일 때만 TRUE를 반환한다.
OR		둘 중 하나만 TRUE이면 TRUE를 반환한다.
=	==	두 항이 같을 경우 TRUE를 반환한다.

- 주석(Comment)

- 주석은 C 프로그래밍 언어에서 사용하는 방법 이외에 두 개의 마이너스 부호(--)를 이용하는 방법이 있다.
- 두 개의 마이너스 부호(--)를 사용하는 주석은 부호(--)가 시작되는 곳에서부터 그 라인의 끝까지 주석으로 처리한다. 또한 EXEC SQL 문장에만 사용될 수 있으며, C 프로그래밍 코드 부분에는 사용되지 못한다.
- 다음은 주석을 사용하는 예이다.

[예 3.2] `tbESQL/C` 프로그램에서의 주석

```
EXEC SQL SELECT ENAME, SALARY, ADDR
        INTO :emp -- 구조체 변수를 사용한다.
        FROM EMP
        WHERE EMPNO =:empno;
```

- 참조(Include)

- `ESQL`에서 헤더 파일을 참조하는 방법으로 `#include` 명령과 EXEC SQL INCLUDE 명령 2가지 방법이 있다.
- 두 명령은 각각 다른 컴파일 단계에서 쓰인다는 점에서 차이를 가지고 있다.

프리컴파일할 때에 필요한 인자들을 참조하기 위해선 EXEC SQL INCLUDE를 통해서 참조해야 하며, 프리컴파일 이후 만들어진 .c 파일을 컴파일할 때 필요한 헤더 파일은 #include를 통해 참조해야 한다. 예를 들어 VARCHAR가 선언되어 있는 헤더 파일은 #include를 통해서 참조할 수 없다.

SQLCA의 경우에는 프리컴파일할 때 자동으로 참조하게 되어 있어서 .c 파일에 남게 된다.

- 다음은 #include와 EXEC SQL INCLUDE를 사용하는 예이다.

[예 3.3] tbESQL/C 프로그램에서의 참조

```
#include <string.h>    /* .c 파일의 컴파일을 위해 필요한 헤더 파일 참조 */
EXEC SQL INCLUDE varchar.h; /* VARCHAR type의 변수가 선언되어 있는 varchar.h를 참조 */
EXEC SQL INCLUDE varchar; /* .h를 생략하고도 사용할 수 있다. */
```

- 주의사항

ESQL 문장 내에서 사용되는 단어 혹은 그 일부를 매크로로 선언하면 프리컴파일 중 Preprocessing 과정에서 단어가 치환되어 ESQL 문장이 해석되지 않을 수 있다. 이를 고려하여 작성해야 한다.

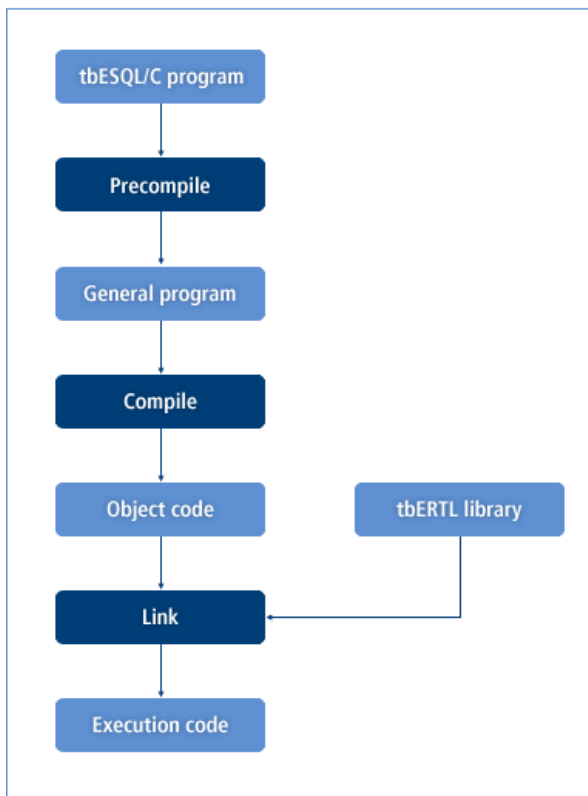
3.1.2. 프로그램의 실행 과정

tbESQL/C 프로그램을 작성할 때는 필요한 데이터 타입이나 함수 프로토타입 등을 이용하기 위해서 반드시 **sqlca.h** 파일을 포함해야 한다. 즉, 아래의 내용이 항상 tbESQL/C 프로그램 소스 코드의 맨 위에 명시되어 있어야 한다. 만약 존재하지 않으면 자동으로 추가된다.

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

다음 그림은 tbESQL/C 프로그램 소스 코드를 실행 파일로 생성하기 위해 거치는 전 과정이다. 프리컴파일 과정을 제외하면 C 프로그램의 경우와 별로 다르지 않다.

[그림 3.1] tbESQL/C 프로그램의 실행 과정



위의 [그림 3.1]의 과정을 순서대로 설명하면 다음과 같다.

1. tbESQL/C program

tbESQL/C 프로그램을 작성한 후 작성된 소스 코드를 저장하면 **.tbc** 확장자를 갖는 파일이 생성된다.

2. Precompile

작성된 프로그램을 실행하려면 먼저 프리컴파일 과정을 거쳐야 한다. tbESQL/C의 프리컴파일러를 실행하는 명령어는 **tbpc**이다.

다음은 emp.tbc 프로그램 파일에 대해 프리컴파일을 실행하는 예이다.

[예 3.4] emp.tbc 프로그램의 프리컴파일

```
$tbpc emp.tbc
```

프리컴파일러를 실행하는 명령어는 옵션을 포함할 수 있다. 다음은 프리컴파일러 옵션을 사용하여 Include 파일의 경로를 지정하는 예이다.

[예 3.5] 프리컴파일러 옵션을 사용한 Include 파일의 경로 지정

```
$tbpc INCLUDE=../include emp.tbc
```

프리컴파일러 옵션에 대한 자세한 내용은 “제9장 tbESQL/C 프리컴파일러 옵션”을 참고한다.

3. General program

프리컴파일의 결과로 C 프로그램 소스 코드가 생성된다. 이때 파일의 이름은 원본 파일의 이름과 동일하고 확장자만 '.c'로 변경된다. 예를 들어 emp.tbc 파일을 프리컴파일하면 emp.c라는 이름을 가진 파일이 생성된다.

4. Compile, Link

프리컴파일이 완료된 파일은 그 다음으로 컴파일 과정과 링크 과정을 거쳐야 한다. [그림 3.1]에서는 컴파일과 링크 과정이 따로 표현되었지만, 실제로는 대개의 경우에 두 과정이 함께 수행된다.

다음은 [예 3.4]의 실행결과로 생성된 emp.c 파일을 컴파일하고 링크하는 예이다.

[예 3.6] 컴파일과 링크 과정

```
$ cc -o emp emp.c -LTB_HOME/client/lib -ltbertl -ltbcli -lpthread -lm
```

tbESQL/C에서는 tbCLI 함수도 함께 사용하기 때문에 tbERTL 라이브러리 이외에 tbCLI 라이브러리를 함께 링크한다. [그림 3.1]에서 링크(Link) 과정에서 링커(Linker)의 입력으로 받아들이는 **tbERTL 라이브러리**는 tbESQL/C의 함수 라이브러리이다. 이 라이브러리에는 esql_do 함수 등이 정의되어 있으며, tbESQL/C 프로그램을 안전하고 효율적으로 실행하기 위한 여러 가지 작업을 수행한다.

5. Execution code

컴파일 과정과 링크 과정을 거치고 나면 실행 파일이 생성된다.

3.1.3. 런타임 에러 처리

tbESQL/C 프로그램 내의 SQL 문장을 실행했을 때 에러 또는 경고 등의 여러 가지 예외 상황이 발생할 수 있다. 예를 들면 SELECT 문장의 실행 결과로 반환되는 로우가 존재하지 않거나 특정 컬럼의 일부 내용이 잘린 경우를 들 수 있다.

tbESQL/C 프로그램 내에서는 에러 또는 경고 상황이 발생한 경우 그에 대한 적절한 처리를 프로그램 내에서 수행할 수 있다.

tbESQL/C에서는 이러한 런타임 에러 처리를 위해 다음의 3가지 인터페이스를 지원한다.

인터페이스	설명
상태 변수	상태 변수(Status Variable)는 임의의 SQL 문장이 실행된 결과가 저장되는 변수이다. 프로그램 내에서는 SQL 문장을 실행한 후에 상태 변수의 값을 검토하여, 에러 또는 경고 상황의 발생을 알 수 있고, 그에 따른 처리를 수행할 수 있다.
SQLCA	SQLCA(SQL 통신 영역 : SQL Communication Area)는 임의의 SQL 문장이 실행된 결과가 저장되는 구조체 변수이다. 이 구조체는 sqlca라는 이름으로 sqlca.h 헤더 파일에 정의되어 있으며, 상태 변수를 포함하고 있다. 상태 변수와 마찬가지로 SQL 문장을 실행한 후에 SQLCA 내의 적절한 멤버 변수의 값을 검토하여 에러 또는 경고 상황의 발생을 알 수 있고, 그에 따른 처리를 수행할 수 있다.
WHENEVER	WHENEVER 문장은 에러 또는 경고 상황이 발생하면 미리 정해진 특정 동작을 수행한다. 상태 변수나 SQLCA 구조체를 이용하면 SQL 문장을 실행할 때마다 에러 또는 경고 상황이 발생하였는지 검토해야 한다. 하지만 WHENEVER 문장을 사용하면 tbESQL/C가 자동으로 예외 상황을 검토하고 그에 따른 처리를 수행한다.

참고

런타임 에러 처리에 대한 자세한 내용은 “제7장 런타임 에러 처리”를 참고한다.

3.2. 프로그램 구조

tbESQL/C 프로그램의 구조는 다음과 같다.

- 변수 선언
- 초기화
- 데이터베이스 작업
- 종료화
- 에러 처리

3.2.1. 변수 선언

변수 선언 부분에는 “1.2. 구성요소”에서 설명한 DECLARE 영역이 포함된다. `tbESQL/C` 문장에서 데이터베이스 작업에 사용될 모든 변수를 DECLARE 영역에 선언해야 한다. 데이터베이스 작업과 관련이 없는 변수는 DECLARE 영역에 포함하지 않아도 된다.

다음은 변수 선언의 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[32];
EXEC SQL END DECLARE SECTION;
```

3.2.2. 초기화

초기화 부분에서는 다음의 두 가지를 수행한다.

- 런타임 에러가 발생했을 때 어떤 작업을 수행할 것인지 선언한다.

런타임 에러가 발생했을 때 수행하는 작업에는 에러 처리 함수를 호출하는 경우, 에러를 무시하고 프로그램을 계속 진행하는 경우, 프로그램을 종료하는 경우, 특정 위치로 이동한 후 실행을 계속하는 경우 등이 있다. 대부분의 경우에 에러 처리를 위한 함수를 미리 정의하고 그 정의된 함수를 호출한다.

다음은 런타임 에러가 발생했을 때 `tbesql_error` 함수를 호출하는 예이다.

[예 3.7] `tbesql_error` 함수 호출

```
EXEC SQL WHENEVER SQLERROR do tbesql_error(...);
```

- Tibero의 데이터베이스에 접속한다.

데이터베이스에 접속할 때는 반드시 사용자 이름과 패스워드를 함께 명시해야 한다.

다음은 두 개의 입력 변수(`username`, `password`)를 이용해 데이터베이스에 접속하는 예이다.

[예 3.8] 입력 변수를 이용해 데이터베이스에 접속

```
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

3.2.3. 데이터베이스 작업

데이터베이스 작업 부분에서는 `tbESQL/C` 문장을 사용해 데이터베이스 질의 및 갱신을 수행한다. 이 부분은 `tbESQL/C` 프로그램에서 가장 중요한 부분 중 하나이다.

데이터베이스와 관련된 작업에는 입력 변수와 출력 변수를 많이 사용하게 된다. 데이터베이스 질의와 관련된 소스 코드에는 커서를 선언하고, 이 선언된 커서를 이용해 로우를 액세스하는 코드가 포함된다.

다음은 데이터베이스 작업 부분의 예이다.

```
EXEC SQL DECLARE C1 CURSOR FOR
    select branch_cd from branch order by branch_cd;
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 INTO :result;
EXEC SQL CLOSE C1;
```

3.2.4. 종료화

종료화 부분에서는 모든 데이터베이스 작업을 마치고 커밋을 수행하거나 롤백을 수행한다.

주의

종료화 부분이 tbESQL/C 프로그램에 포함되지 않으면, 자동으로 커밋되지 않으므로 주의한다.

다음은 데이터베이스에 부분 롤백을 수행한 뒤 커밋을 하는 예이다.

```
EXEC SQL ROLLBACK WORK TO SAVEPOINT SP1;
EXEC SQL COMMIT WORK;
```

3.2.5. 에러 처리

에러 처리 부분에서는 런타임 에러를 처리하기 위한 코드가 포함된다. 에러 처리와 관련된 코드는 다른 코드와 섞여 동일한 하나의 함수 안에 포함될 수도 있으며, 별도의 함수로 정의할 수도 있다.

다음은 에러 처리의 예이다.

```
EXEC SQL WHENEVER SQLERROR
DO printf("file: %s, line: %d, err_code: %d\n", __FILE__, __LINE__, sqlca.sqlcode);
```

3.3. tbESQL/C 문장 실행

본 절에서는 tbESQL/C 프로그램에서 SELECT, INSERT, UPDATE, DELETE 문장을 실행하는 방법에 대해 설명한다.

각 문장에는 입/출력 변수가 사용되는데, 입/출력 변수는 각 문장 내에 포함된 스키마 객체와 구별하기 위하여 반드시 앞에 **콜론(:)**이 와야 한다. 입/출력 변수는 이미 각 용도에 맞게 선언된 C 프로그래밍 언어의 변수이며, 이 변수를 **호스트 변수**라고 한다.

3.3.1. SELECT

SELECT 문장은 데이터베이스에 질의를 수행하고 결과 로우를 반환하는 문장이다. 결과 로우의 개수는 보통 하나 이상이지만 하나도 없을 수도 있다.

INTO 절과 출력 변수

tbESQL/C 프로그램 내에서 사용되는 **SELECT** 문장은 일반 **SELECT** 문장과 같은 문법을 가진다. 다만 **SELECT** 리스트 다음에 결과 로우의 각 컬럼 값을 **출력 변수**에 저장하기 위해 **INTO 절**이 삽입된다.

다음은 **SELECT** 문장의 **INTO 절**에 출력 변수가 사용된 예이다.

```
EXEC SQL SELECT LENGTH(VARCHAR_COL)
          INTO :col_len FROM TEST;
```

다음은 **SELECT** 문장의 **INTO 절**에 포함되는 출력 변수에 대한 설명이다.

- 컬럼 값과 출력 변수의 대응

INTO 절에 포함되는 출력 변수는 **SELECT** 리스트 내의 컬럼과 같은 개수이어야 하며, 지시자 변수와 함께 사용될 수 있다. 질의 결과로 반환된 로우의 각 컬럼 값은 컬럼 값과 동일한 순서로 대응되는 각각의 출력 변수에 저장된다. 출력 변수에 저장될 때 **tbESQL/C** 프로그램에서는 필요한 경우 데이터 타입의 변환을 수행한다.

- 구조체 변수

INTO 절에 포함되는 출력 변수에는 구조체 변수를 사용할 수도 있다. 이때 구조체 변수에 포함된 멤버 변수의 개수는 **SELECT** 리스트 내의 컬럼과 개수가 같아야 한다. **tbESQL/C** 프로그램에서는 결과 로우의 각 컬럼 값을 구조체 변수 내의 각 멤버 변수에 할당한다.

- 로우의 개수에 따른 출력 변수

SELECT 문장의 결과 로우의 개수가 반드시 하나라는 보장이 있다면 **INTO 절**에 단순 출력 변수를 이용하여 처리가 가능하다. 하지만 하나 이상인 경우에는 커서를 사용하거나 **INTO 절**에 출력 배열 변수를 사용해야 한다.

참고

커서에 대한 자세한 내용은 “[3.4. 커서](#)”와 “[3.5. 스크롤 가능 커서](#)”, 배열 변수에 대한 자세한 내용은 “[제4장 배열 변수](#)”를 참고한다.

다음은 결과 로우가 하나인 경우 이를 처리하는 예이다.

```
VARCHAR ename[24];
int salary;
int empno;
```

```
...
empno = 20;
EXEC SQL SELECT ENAME, SALARY * 1.05
      INTO :ename, :salary
      FROM EMP
      WHERE EMPNO = :empno;
printf("ename = %.*s\n", ename.len, ename.arr);
```

위의 예에서 컬럼 EMPNO가 테이블 EMP의 기본 키 컬럼이므로 질의 결과 로우의 개수가 하나라는 것을 알 수 있다.

WHERE 절과 입력 변수

SELECT 문장 내에서 변수의 위치는 INTO 절에 출력 변수가 포함되는 것 외에 **WHERE 절에 입력 변수**가 포함된다.

다음은 SELECT 문장의 WHERE 절에 입력 변수가 사용된 예이다.

```
EXEC SQL SELECT COUNT(*) INTO :cnt FROM TEST
      WHERE VARCHAR_COL = :varchar_col;
```

다음은 SELECT 문장의 WHERE 절에 포함되는 입력 변수에 대한 설명이다.

- 입력 변수의 값 설정

WHERE 절에 포함되는 입력 변수의 값은 SELECT 문장이 실행되기 전에 설정되어 있어야 한다. 그 이유는 SELECT 문장은 실행 직전에 입력 변수의 값을 가져와서 SELECT 문장을 완성한 뒤에 실행되기 때문이다. SELECT 문장의 실행에 필요한 입력 변수의 값은 그 문장이 실행되기 직전에 읽혀지므로, 입력 변수의 값은 프로그램 실행 중에 동적으로 설정할 수 있다.

- 입력 변수 사용의 제약

SELECT 문장의 입력 변수는 상수를 대신하여 사용할 수 있지만 스키마 객체 또는 컬럼 등의 이름을 대신하여 사용될 수는 없다. SELECT 문장에 부질의(Subquery)가 사용되는 경우에는 부질의 내에 출력 변수를 포함시킬 수는 없지만 입력 변수를 포함시킬 수는 있다.

다음은 컬럼의 이름을 대신하여 입력 변수가 사용된 예이다.

```
VARCHAR ename[24];
int salary;
VARCHAR col_name[32];
...
strcpy(col_name.arr, "EMPNO");
col_name.len = strlen(col_name.arr);
EXEC SQL SELECT ENAME, SALARY
```

```

    INTO :ename, :salary
  FROM EMP
  WHERE :col_name = 20;

```

위의 예의 맨 마지막 라인에서 **WHERE** 절 다음에 컬럼의 이름이 나와야 하는데, 대신 변수가 사용되었기 때문에 잘못되었다.

3.3.2. INSERT

INSERT, DELETE, UPDATE 문장은 공통적으로 질의의 결과 로우가 존재하지 않으므로 출력 변수 없이 입력 변수만을 사용한다. **INSERT** 문장에서 입력 변수는 **컬럼**에 삽입할 데이터 값의 위치나 **부질의** 내부에 사용될 수 있다.

INSERT 문장에서 삽입하고자 하는 컬럼 값의 일부에 대해서만 입력 변수를 사용할 수도 있다.

다음은 일부 컬럼에 대해서만 입력 변수를 사용하는 예이다.

```

VARCHAR ename[24];
int empno;
...
strcpy(ename.arr, "Michael");
empno = 25;
EXEC SQL INSERT INTO EMP (ENAME, SALARY, EMPNO)
      VALUES(:ename, 35000, :empno);

```

삽입하고자 하는 모든 컬럼 값에 대하여 입력 변수를 사용하는 경우 구조체 변수를 사용할 수 있다. 이때 구조체 변수에 포함된 각 변수 값은 삽입하고자 하는 각 컬럼 값에 대응된다.

다음은 구조체 변수를 사용해 데이터를 삽입하는 예이다.

```

struct {
    VARCHAR ename[24];
    int salary;
    int empno;
} emp;
...
strcpy(emp.ename.arr, "Michael");
emp.ename.len = strlen(emp.ename.arr);
emp.salary = 35000;
emp.empno = 25;
EXEC SQL INSERT INTO EMP (ENAME, SALARY, EMPNO)
      VALUES (:emp);

```

또한 부질의에 입력 변수를 사용할 수도 있다. 다음은 부질의를 포함하는 INSERT 문장의 예이다.

```

int sal_bound;
...
sal_bound= 30000;
EXEC SQL INSERT INTO EMP_SUB (ENAME, EMPNO)
      SELECT ENAME, EMPNO FROM EMP
      WHERE SALARY >= :sal_bound;

```

3.3.3. UPDATE

UPDATE 문장도 INSERT 문장과 마찬가지로 입력 변수만을 사용한다.

UPDATE 문장에서는 **SET 절**의 컬럼 값의 위치나 **WHERE 절**에서 입력 변수가 사용될 수 있다. UPDATE 문장에 포함된 **부질의**에서도 입력 변수를 사용할 수 있다. UPDATE 문장에서는 INSERT 문장에서와 달리 구조체 입력 변수를 사용할 수 없다.

다음은 일부 컬럼 값과 WHERE 절 내에 입력 변수를 사용하는 예이다.

```

VARCHAR ename[24];
int empno;
...
EXEC SQL UPDATE EMP
      SET ENAME = :ename.arr, SALARY = SALARY * 1.05
      WHERE EMPNO = :empno;

```

다음은 부질의에 입력 변수를 사용하는 예이다.

```

int empno;
int sal_bound;
...
empno = 20;
sal_bound = 30000;
EXEC SQL UPDATE EMP
      SET SALARY =(SELECT SALARY FROM EMP WHERE EMPNO = :empno)
      WHERE SALARY <=:sal_bound;

```

3.3.4. DELETE

DELETE 문장도 INSERT, UPDATE 문장과 마찬가지로 입력 변수만을 사용한다. DELETE 문장에서는 **WHERE 절**에서 입력 변수가 사용된다. DELETE 문장에서도 UPDATE 문장과 마찬가지로 구조체 입력 변수를 사용할 수 없다.

다음은 입력 변수를 사용하는 DELETE 문장의 예이다.

```
int sal_bound;
...
sal_bound = 25000;
EXEC SQL DELETE FROM EMP
        WHERE SALARY <= :sal_bound;
```

3.4. 커서

본 절에서는 커서의 기본적인 사용 방법에 대하여 설명하고, 갱신 및 삭제를 위한 **CURRENT OF** 절을 설명한다. 그리고 마지막으로 사용 예제를 제시한다.

3.4.1. 사용 방법

SELECT 문을 통한 질의를 수행할 때 **WHERE** 절에 기본 키 제약조건을 부여하지 않으면, 대개의 경우 결과 로우의 개수는 하나 이상이다. **커서**는 이렇게 반환된 다수의 결과 로우에 각각 차례로 액세스하기 위한 데이터 구조이다.

다음은 커서를 사용하는 순서이다.

1. 커서를 사용하기 위해서는 **DECLARE CURSOR**를 사용해 맨 먼저 **SQL** 문장과 연관하여 커서를 선언해야 한다. 커서를 선언할 때에는 항상 커서의 이름을 주어야 하며, 커서의 선언부는 그 커서를 사용하는 다른 모든 문장의 앞에 와야 한다.

다음은 emp_cursor라는 이름으로 커서를 선언하는 예이다.

[예 3.9] 커서의 선언

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT ENAME, SALARY, ADDR
        FROM EMP
        WHERE DEPTNO = :deptno;
```

2. 커서를 사용하기 위해서는 **OPEN**을 사용해 해당 커서를 열어야 한다.

OPEN을 실행하면 연관된 **SELECT** 문장이 실행되어 질의의 결과 로우가 반환된다. 또한 커서는 결과 로우 중에서 맨 처음에 위치한 로우의 직전을 가리킨다. 첫 **FETCH**가 실행되면 첫 번째 로우를 가리키게 된다. **OPEN을 실행해서 커서를 선언할 때에 SELECT 문장에 포함된 입력 변수의 값은 OPEN이 실행될 때 할당된다는 것에 유의한다.**

다음은 [예 3.9]에서 선언한 emp_cursor라는 이름의 커서에 **OPEN**을 실행하는 예이다.

[예 3.10] OPEN의 실행

```
EXEC SQL OPEN emp_cursor;
```

3. **FETCH**를 실행해 로우에 액세스를 한다.

OPEN의 실행으로는 아직 로우에 액세스를 할 수 있는 것은 아니다. 로우에 액세스를 하기 위해서는 FETCH를 실행해야 한다. FETCH의 INTO 절에는 구조체 변수나 지시자 변수를 함께 사용할 수 있다.

다음은 FETCH를 실행하는 예이다.

[예 3.11] FETCH의 실행

```
EXEC SQL FETCH emp_cursor
      INTO :ename, :salary, :addr;
```

FETCH를 실행하면 먼저 커서가 다음 결과 로우를 가리키게 되고, 커서가 가리키는 결과 로우를 출력 변수에 저장한다. FETCH를 실행할 때마다 커서는 다음 결과 로우를 가리키고, 결국 맨 마지막 결과 로우의 범위를 넘어 FETCH를 실행하면, NOT FOUND 에러가 발생한다.

대개의 경우 FETCH를 **무한 루프** 안에 포함시키며, **NOT FOUND 에러**가 발행하면 루프를 빠져 나오도록 코드를 작성한다. 이때 NOT FOUND 에러가 발생했을 때 루프를 빠져 나오도록 하기 위해서는 **WHENEVER** 문장을 사용한다.

다음은 WHENEVER 문장을 사용하는 예이다.

[예 3.12] WHENEVER 문장의 사용

```
EXEC SQL WHENEVER NOT FOUND DO break;
while (1) {
    EXEC SQL FETCH emp_cursor
        INTO :ename, :salary, :addr;
    ...
}
```

WHENEVER 문장에 DO break 명령 이외에 GOTO 명령을 사용할 수도 있다.

OPEN을 사용해 커서를 열기 전이나 CLOSE를 사용해 커서를 닫은 후 그리고 NOT FOUND 에러가 발생한 이후에 FETCH를 실행하면 에러가 발생한다.

FETCH를 이용해 다음 로우 뿐만 아니라 이전 로우를 액세스할 수도 있다. 이러한 작업을 위해서는 스크롤 가능 커서를 선언해야 한다. 스크롤 가능 커서에 대해서는 “3.5. 스크롤 가능 커서”를 참고한다.

4. 커서 사용의 마지막 단계는 CLOSE를 사용해 커서를 닫는 것이다. 커서를 닫은 이후에는 그 커서에 대해 어떠한 작업도 실행할 수 없다.

다음은 emp_cursor라는 이름의 커서에 CLOSE를 실행하는 예이다.

[예 3.13] CLOSE 사용 예

```
EXEC SQL CLOSE emp_cursor;
```

3.4.2. CURRENT OF 절

커서를 이용해 SELECT 문장의 실행 결과 로우를 차례로 액세스하면서 커서가 현재 가리키고 있는 결과 로우를 삭제하거나 갱신하려고 할 때 DELETE 문장과 UPDATE 문장에 **CURRENT OF 절**을 사용한다.

SELECT 문장에서 CURRENT OF 절을 사용하기 위해서는 FOR UPDATE 절을 포함해야 한다. **FOR UPDATE 절**은 질의 결과로 반환된 로우에 잠금(LOCK)을 설정한다. 잠금이 설정된 로우는 현재 트랜잭션이 커밋 또는 롤백되기 전까지는 다른 트랜잭션이 로우를 갱신하거나 삭제할 수 없다.

다음은 UPDATE 문장에서 CURRENT OF 절을 이용하여 컬럼 SALARY만을 갱신하는 예이다.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, SALARY, ADDR
    FROM EMP
    WHERE DEPTNO =:deptno
    FOR UPDATE OF SALARY;

...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO break;

while (1) {
    EXEC SQL FETCH emp_cursor
        INTO :ename, :salary, :addr;

    ...
    EXEC SQL UPDATE EMP
        SET SALARY = SALARY * 1.05
        WHERE CURRENT OF emp_cursor;

    ...
}
```

커서가 현재 가리키고 있는 로우는 FETCH 문장을 실행하여 방금 전에 컬럼 값을 읽은 로우이다.

커서에 대해 OPEN 문장을 실행한 후에 한번도 FETCH 문장을 실행하지 않았거나 모든 결과 로우를 읽고 나서 NOT FOUND 에러가 반환되었다면 커서가 현재 가리키고 있는 로우는 없다. 현재 가리키고 있는 로우가 없는 커서를 이용하여 DELETE 또는 UPDATE 문장을 실행하였다면 에러를 반환한다. 또한 OPEN 문장을 수행하지 않았거나 CLOSE 문장을 이미 수행한 커서를 이용하여 삭제 또는 갱신을 시도할 때에도 에러를 반환한다.

CLOSE_ON_COMMIT 옵션이 'YES'로 지정된 경우를 제외하고는 일반적으로 커서는 현재 트랜잭션이 커밋 또는 롤백한 후에도 사용할 수 있다. 즉, 커서를 이용하여 질의 결과 로우를 액세스할 수 있다. 하지만 FOR UPDATE 절을 포함한 SELECT 문장에 대한 커서는 사용할 수 없다. 왜냐하면 트랜잭션이 커밋되거나 롤백되는 동시에 결과 로우에 설정되었던 잠금을 해제해 버리기 때문이다.

3.4.3. 사용 예제

다음은 커서를 사용하는 예제 프로그램이다.

[예 3.14] 커서의 사용

```
#include <stdio.h>
#include <sqlca.h>
#include <string.h>
#define USERPASS "tibero/tibero"
int main()
{
    EXEC SQL BEGIN DECLARE SECTION;                                ... ① ...
        VARCHAR userpass[20] = {strlen(USERPASS), USERPASS};
        VARCHAR ename[24];
        int salary;
        VARCHAR addr[32];
        int deptno;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL DECLARE emp_cursor CURSOR FOR                          ... ② ...
        SELECT ENAME, SALARY, ADDR
        FROM EMP
        WHERE DEPTNO =:deptno;                                       ... ③ ...

    EXEC SQL DECLARE emp_update_cursor CURSOR FOR                  ... ④ ...
        SELECT SALARY
        FROM EMP
        WHERE DEPTNO =:deptno                                       ... ⑤ ...
        FOR UPDATE OF SALARY;

    EXEC SQL CONNECT :userpass;

    printf("Connected.\n");
    printf("Enter dept number to show: ");
    scanf("%d", &deptno);

    EXEC SQL OPEN emp_cursor;
    EXEC SQL WHENEVER NOT FOUND DO break;

    while (1)
    {
        EXEC SQL FETCH emp_cursor
            INTO :ename, :salary, :addr;

        printf("ename = %.*s, salary = %d, addr= %.*s\n",
            ename.len, ename.arr, salary, addr.len, addr.arr);
    }
}
```

```

}

EXEC SQL CLOSE emp_cursor;

printf("Enter dept number to raise salary: ");
scanf("%d",&deptno);

EXEC SQL OPEN emp_update_cursor;
EXEC SQL WHENEVER NOT FOUND DO break;

while (1)
{
    EXEC SQL FETCH emp_update_cursor
        INTO :salary;

    EXEC SQL UPDATE EMP
        SET SALARY = :salary * 1.05
        WHERE CURRENT OF emp_update_cursor;
}

EXEC SQL CLOSE emp_update_cursor;
EXEC SQL COMMIT WORK RELEASE;                ... ⑥ ...

return 1;
}

```

① **tbESQL/C** 문장 내에 포함되는 모든 입/출력 변수는 **DECLARE** 영역 안에서 선언한다. 프로그램의 맨 앞쪽에서 커서를 선언하고 있지만, 변수의 선언과는 달리 커서의 선언은 어떠한 위치에 오더라도 상관없으며, 그 커서가 사용되기 전에만 선언되면 된다.

②, ④ 두 개의 커서를 선언한다. 각각 단순 질의와 갱신을 위한 커서인데, 단순 질의를 위한 커서는 **emp_cursor**이고, 갱신을 위한 커서는 **emp_update_cursor**이다.

③, ⑤ 변수 **deptno**가 두 개의 커서에 공통적으로 사용된다. 커서와 연관된 **SELECT** 문장은 **OPEN** 문장으로 커서를 열 때 실행되며, 그 직전에 입력 변수의 값을 읽어 들인다. 따라서 같은 변수를 사용하더라도 각각의 **SELECT** 문장이 실행될 때 서로 다른 **deptno** 값이 적용될 수도 있다.

⑥ 프로그램의 맨 마지막에서는 현재 트랜잭션을 커밋한다. 단순 질의를 위한 커서 **emp_cursor**는 트랜잭션 커밋 후에도 계속 사용할 수 있으나, 갱신을 위한 커서 **emp_update_cursor**는 사용할 수 없다.

3.5. 스크롤 가능 커서

본 절에서는 스크롤 가능 커서(Scrollable Cursors)의 사용 방법과 예제를 설명한다.

3.5.1. 사용 방법

커서는 질의의 결과 로우를 액세스할 때 항상 다음에 위치한 로우만 액세스할 수 있는 것에 비해, **스크롤 가능 커서**는 임의의 로우에 액세스를 할 수 있다. 예를 들어 스크롤 가능 커서는 현재 커서가 가리키고 있는 로우의 바로 이전 로우를 액세스하거나 전체 결과 로우 중에서 n번째 로우를 액세스할 수 있다.

스크롤 가능 커서는 사용 방법의 편리성과 유연성을 제공하지만 커서와 비교했을 때 메모리 등의 리소스를 많이 사용할 수 있으므로 프로그램의 실행 성능을 떨어뜨릴 수 있다. 따라서 꼭 필요한 경우가 아니라면 커서를 사용하는 것이 효율적이다.

다음은 커서와 스크롤 가능 커서의 차이점이다.

커서	스크롤 가능 커서
다음 위치에 있는 로우만 차례대로 액세스를 한다.	임의의 위치에 있는 로우를 액세스할 수 있다.
'DECLARE {커서 이름} CURSOR'의 형태로 선언한다.	'DECLARE {커서 이름} SCROLL CURSOR'의 형태로 선언한다.
FETCH를 실행할 때 옵션을 지정할 수 없다.	FETCH를 실행할 때 반드시 옵션을 지정해야 한다.

스크롤 가능 커서에서도 커서와 동일하게 OPEN과 CLOSE를 사용한다. 스크롤 가능 커서가 현재 가리키고 있는 로우에 대하여 삭제 및 갱신을 수행하고자 할 때에도 커서와 마찬가지로 DELETE 문장과 UPDATE 문장 내에서 CURRENT OF 절을 이용한다. 문장의 작성 및 사용 방법은 커서와 동일하다.

커서 이름을 지정하는데는 다음 사항을 고려해야 한다.

- 커서 이름에 "SQLCUR", "SQL_CUR"를 포함하지 않아야 한다.
- 커서 이름의 길이가 128이 넘어가지 않도록 한다.
- 커서 이름의 길이가 128이 넘어가면 CLI단에서 이름을 128로 자른다.

커서와 스크롤 가능 커서의 차이점을 좀더 상세하게 설명하면 다음과 같다.

• 스크롤 가능 커서의 선언

스크롤 가능 커서의 선언은 **SCROLL** 키워드가 포함된다는 것을 제외하면 커서의 선언과 동일하며 다음과 같은 형태로 선언한다.

```
DECLARE {커서 이름} SCROLL CURSOR
```

다음의 소스 코드는 스크롤 가능 커서를 선언하는 예이다.

```
EXEC SQL DECLARE emp_scroll_cursor SCROLL CURSOR FOR
SELECT ENAME, SALARY, ADDR
```

```
FROM EMP
WHERE DEPTNO = :deptno;
```

• 스크롤 가능 커서에서의 FETCH의 사용

스크롤 가능 커서에서 **FETCH**를 사용할 때는 항상 액세스할 대상 로우를 지정해야 한다.

다음은 **FETCH**에서 액세스를 할 대상 로우를 지정할 때 사용되는 옵션이다.

옵션	설명
NEXT	현재 커서가 가리키는 로우의 다음 로우에 액세스를 한다. PRIOR 옵션과 반대이다. (생략 가능)
PRIOR	현재 커서가 가리키는 로우의 이전 로우에 액세스를 한다. NEXT 옵션과 반대이다.
FIRST	맨 처음에 위치한 로우에 액세스를 한다. LAST 옵션과 반대이다.
LAST	맨 마지막에 위치한 로우에 액세스를 한다. FIRST 옵션과 반대이다.
CURRENT	현재 로우에 액세스를 한다.
RELATIVE offset	현재 커서가 가리키고 있는 로우의 다음 offset 번째에 위치한 로우에 액세스를 한다. offset 값이 음수라면 커서가 현재 위치에서 앞으로 이동한다. 예를 들어 현재 커서가 8번째 로우를 가리키고 있는데, ' FETCH RELATIVE -3 '을 실행한다면 커서는 5번째 로우를 가리키게 된다.
ABSOLUTE offset	전체 로우 중에서 offset 번째 로우에 액세스를 한다.

다음은 **FETCH**에 옵션을 사용하는 예이다.

```
EXEC SQL FETCH PRIOR emp_scroll_cursor
      INTO :ename, :salary, :addr;

EXEC SQL FETCH LAST emp_scroll_cursor
      INTO :ename, :salary, :addr;

EXEC SQL FETCH ABSOLUTE 3 emp_scroll_cursor
      INTO :ename, :salary, :addr;
```

위의 예에서는 각각 순서대로 첫 번째 문장은 이전 로우를 액세스하고, 두 번째 문장은 마지막 로우를 액세스하고, 세 번째 문장은 전체 로우 중에서 세 번째 로우를 액세스하고 있다. 만약 액세스하고자 하는 로우가 존재하지 않으면 **NOT FOUND** 에러가 반환된다.

3.5.2. 사용 예제

다음은 스크롤 가능 커서를 사용하는 예제 프로그램이다.

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

```

#include <string.h>
#define USERPASS "tibero/tibero"
int main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userpass[20]= { strlen(USERPASS), USERPASS };
    VARCHAR ename[24]; int salary;
    VARCHAR addr[32];
    int deptno;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL DECLARE emp_scroll_cursor SCROLL CURSOR FOR
        SELECT ENAME, SALARY, ADDR FROM EMP
        WHERE DEPTNO = :deptno;

    EXEC SQL CONNECT :userpass;
    printf("Connected.\n");
    printf("Enter dept number to show: ");
    scanf("%d", &deptno);

    EXEC SQL OPEN emp_scroll_cursor;

    EXEC SQL FETCH FIRST emp_scroll_cursor
        INTO :ename, :salary, :addr; /* 1st row */

    EXEC SQL FETCH LAST emp_scroll_cursor
        INTO :ename, :salary, :addr; /* last row */

    EXEC SQL FETCH ABSOLUTE 5 emp_scroll_cursor
        INTO :ename, :salary, :addr; /* 5th row */

    EXEC SQL FETCH RELATIVE 3 emp_scroll_cursor
        INTO :ename, :salary, :addr; /* 8th row */

    EXEC SQL FETCH PRIOR emp_scroll_cursor
        INTO :ename, :salary, :addr; /* 7th row */

    EXEC SQL FETCH CURRENT emp_scroll_cursor
        INTO :ename, :salary, :addr; /* 7th row */

    EXEC SQL FETCH RELATIVE -3 emp_scroll_cursor
        INTO :ename, :salary, :addr; /* 4th row */

    EXEC SQL FETCH emp_scroll_cursor
        INTO :ename, :salary, :addr; /* 5th row */

    EXEC SQL CLOSE emp_scroll_cursor;

```

```
EXEC SQL COMMIT WORK RELEASE;  
  
    return 1;  
}
```

위의 예에서는 주석을 삽입하여 **FETCH**를 실행할 때마다 스크롤 가능 커서의 현재 위치를 설명하였다. 또한 질의 결과 로우가 8개 이상임을 가정한다. **SQL** 문장 내에 포함되는 모든 입/출력 변수는 **DECLARE** 영역 안에서 선언하였다. 커서를 선언할 때와 마찬가지로 스크롤 가능 커서를 선언하는 문장도, 스크롤 가능 커서가 사용되기 전이라면 어떤 위치에 있어도 상관없다.

제4장 배열 변수

본 장에서는 배열 변수의 기본 개념과 선언 방법, 그리고 배열 변수를 `tbESQL/C` 문장에서 입/출력 변수로 사용하는 방법을 설명한다.

4.1. 개요

`tbESQL/C` 프로그램의 배열 변수는 `C` 프로그래밍 언어의 배열 변수의 개념과 동일하다. 배열은 동일한 타입의 값을 여러 개 저장할 수 있는 데이터 구조이다.

`tbESQL/C` 프로그램에서는 `SELECT` 문장을 실행한 뒤 다수의 결과 로우를 저장하기 위해서 출력 변수로 배열 변수를 사용하거나, `INSERT` 문장을 사용할 때도 보통 여러 개의 로우를 삽입하게 되므로 입력 변수로 배열 변수를 사용한다.

배열 변수가 출력 변수로 사용될 때는 **출력 배열 변수**라고 부르며, 입력 변수로 사용될 때는 **입력 배열 변수**라고 한다. 출력 배열 변수와 입력 배열 변수가 사용되는 문장은 다음과 같다.

- 출력 배열 변수
 - `SELECT`
- 입력 배열 변수
 - `INSERT`
 - `UPDATE`
 - `DELETE`

`tbESQL/C` 프로그램에서 `tbESQL/C` 문장과 `C` 프로그램의 변수 사이에 데이터를 주고 받을 때 배열 변수를 사용하면 다음과 같은 장점이 있다.

- 간결하고 구조적인 프로그래밍

배열 변수를 사용하면 각각의 로우를 처리할 때마다 별도의 `SQL` 문장을 사용하지 않고, 모든 로우를 하나의 `SQL` 문장으로 처리할 수 있다. 따라서 프로그램의 소스 코드를 보다 단순화할 수 있으며, 구조적인 프로그래밍이 가능하다.

- `tbESQL/C` 프로그램의 성능 향상

클라이언트와 서버 환경에서는 데이터를 주고받기 위하여 많은 시간이 필요하다. 배열 변수를 사용해 여러 로우를 한꺼번에 처리하면 전송 시간이 크게 줄어든다.

4.2. 배열 변수 선언

tbESQL/C 문장에서 사용될 배열 변수를 선언할 때에는 다음의 내용에 유의해야 한다.

- 반드시 DECLARE 영역 안에 선언해야 한다.
- VARCHAR 또는 CHAR 타입 등의 문자열 타입 변수는 항상 2차원 배열이어야 한다.
- int 타입 등의 문자열이 아닌 변수는 항상 1차원 배열이어야 한다.
- 포인터 변수를 배열 변수 형태로 선언하면, tbESQL/C 문장에서 사용할 수 없다.

다음은 배열 변수를 선언하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;  
VARCHAR ename[50][24];  
int salary[50];  
VARCHAR addr[50][32];  
...  
EXEC SQL END DECLARE SECTION;
```

다음은 배열 변수를 잘못 선언한 예이다.

```
VARCHAR ename[50][20][24];      ... ① ...  
int salary[50][20];             ... ② ...  
int *salary[50];  
...  
EXEC SQL SELECT SALARY, ...  
      INTO :salary, ...         ... ③ ...  
      FROM EMP;
```

- ① VARCHAR 타입 변수는 2차원으로 선언되어야 하는데 3차원으로 잘못 선언된 예이다.
- ② int 타입 등의 문자열이 아닌 변수는 1차원으로 선언되어야 하는데 2차원으로 잘못 선언된 예이다.
- ③ 포인터 변수를 배열 변수로 선언해서 tbESQL/C 문장에 사용하려고 했기 때문에 잘못 선언된 예이다.

4.3. 입/출력 배열 변수

본 절에서는 tbESQL/C 문장에서 배열 변수가 입/출력 변수로 사용될 경우 각 tbESQL/C 문장에 따른 배열 변수의 사용 방법에 대해 설명한다.

4.3.1. SELECT

SELECT 문장을 실행하여 반환되는 결과 로우의 개수가 하나 이상이라면 반드시 커서 또는 **배열 변수**를 사용해야 한다. 결과 로우의 개수를 미리 예측할 수 있다면 배열 변수의 크기를 충분히 선언하여 사용할

수 있다. 만약 결과 로우의 개수가 예측하기 힘들거나 매우 크다면, **배열 변수와 커서를 함께** 사용할 수도 있다.

배열 변수

SELECT 문장의 실행 결과로 반환된 로우의 개수가 예측 가능하고 크기가 크지 않다면 **배열 변수**를 이용하여 모든 결과 로우를 한 번에 받을 수 있다.

예를 들어 **SELECT** 문장의 결과로 반환되는 로우의 개수가 50개를 넘지 않는 경우에 다음과 같이 소스 코드를 작성할 수 있다.

[예 4.1] SELECT 문장에서의 배열 변수의 사용

```
char ename[50][24];
int salary[50];
VARCHAR addr[50][32];
...
EXEC SQL SELECT ENAME, SALARY, ADDR
      INTO :ename, :salary, :addr
      FROM EMP
      WHERE SALARY >= 50000;
```

결과 로우의 각 컬럼 값은 각 배열 변수에 저장되며, 저장될 때 같은 순서의 변수에 저장된다. 예를 들어 세 번째 로우의 컬럼 값은 각각 배열 변수의 세 번째 위치인 `ename[2]`, `salary[2]`, `addr[2]`에 저장된다.

SELECT 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- INTO 절에 포함되는 출력 변수는 배열 변수와 일반 변수가 동시에 **함께 올 수 없다**. 따라서 INTO 절에 포함된 모든 출력 변수는 배열 변수이거나 또는 일반 변수이어야 한다.
- **SELECT** 문장의 실행 결과로 반환된 로우의 전부가 아닌 **일부**만을 얻고 싶을 때에도 배열 변수를 이용할 수 있다. 위의 [예 4.1]에서 **SELECT** 문장의 실행 결과로 반환된 로우가 50개가 넘는 경우에 50개까지의 결과 로우만을 배열 변수에 저장한다. 만약 결과로 반환된 로우를 30개까지만 저장하고 싶다면, 위의 [예 4.1]의 배열 변수를 다음과 같이 선언하면 된다.

```
char ename[30][24];
int salary[30];
VARCHAR addr[30][32];
```

- **SELECT** 문장의 실행 결과로 배열 변수에 저장된 로우의 실제 개수는 SQLCA의 변수인 **sqlca**를 통해 알 수 있다. 변수 **sqlca**는 **tbESQL/C** 프로그램 내에 포함된 변수로 주로 에러 및 경고 처리에 사용된다. **sqlca**의 사용에 대해서는 “7.3.1. SQLCA 구조체 변수”를 참고한다.

정확하게는 **sqlca.sqlerrd[2]**에 실제로 변수에 저장된 로우의 개수가 저장된다. 다음은 **SELECT** 문의 실행 후에 올 수 있는 소스 코드에 **sqlca.sqlerrd[2]**를 사용한 예이다.

[예 4.2] sqlca.sqlerrd[2]의 활용

```
for (i = 0; i < sqlca.sqlerrd[2]; i++)
{
    printf("%s, %d, %.*s\n", ename[i], salary[i], addr[i].len, addr[i].arr);
    ...
}
printf("number of returned rows = %d\n", sqlca.sqlerrd[2]);
```

맨 마지막 라인을 보면 반환된 로우의 개수를 출력하는 데 sqlca.sqlerrd[2]를 사용하였다.

- 배열 변수의 크기가 일정하지 않은 경우에는 **가장 작은** 배열 변수의 크기를 전체 배열 변수의 크기로 설정한다. 예를 들어 다음의 소스 코드와 같이 배열 변수가 선언된 경우에 배열 변수 ename과 salary의 크기는 50이지만, addr의 크기가 30이므로 SELECT 문의 실행 후에 반환되는 결과 로우의 개수는 30개이다.

```
char ename[50][24];
int salary[50];
VARCHAR addr[30][32];
...
EXEC SQL SELECT ENAME, SALARY, ADDR
        INTO :ename, :salary, :addr
        FROM EMP
        WHERE SALARY >= 50000;
```

- SELECT 문장의 **WHERE** 절에는 배열 변수가 올 수 없다. WHERE 절에 배열 변수가 오는 경우 질의의 의미가 모호해지기 때문이다. 따라서 다음과 같은 소스 코드는 프리컴파일 과정에서 에러를 반환한다.

```
int deptno[50];
...
EXEC SQL SELECT ENAME, SALARY, ADDR
        INTO :ename, :salary, :addr
        FROM EMP
        WHERE DEPTNO = :deptno;
```

배열 변수와 함께 커서를 사용하더라도 SELECT 문장의 WHERE 절에 배열 변수를 사용할 수는 없다.

배열 변수와 커서

SELECT 문장의 실행 결과로 반환되는 로우의 개수를 예측하기 어렵거나 반환되는 로우의 개수가 많다면 배열 변수와 함께 **커서**를 이용해야 한다. 이때 일반 커서는 물론 스크롤 가능 커서도 사용할 수 있다.

커서와 배열 변수를 함께 사용하는 방법은 커서와 일반 변수를 함께 사용하는 경우와 거의 유사하다. 하지만 **FETCH**를 실행할 때 루프를 빠져 나오는 방법이 일반 변수를 사용할 때와 다르다. 그 이유는 NOT FOUND 에러가 발생하는 경우가 차이가 있기 때문이다.

일반적으로 커서를 이용하여 루프 내에서 결과 로우를 액세스할 때 더 이상 읽어 올 결과 로우가 없으면 NOT FOUND 에러가 반환된다. 하지만 일반 변수일 때와 배열 변수일 때 **NOT FOUND 에러**는 다음과 같은 차이가 있다.

구분	설명
일반 변수	일반 변수를 이용할 때에는 결과 로우를 하나씩 액세스하므로 NOT FOUND 에러가 발생할 때에는 출력 변수에 저장된 결과 로우는 없다.
배열 변수	배열 변수를 이용할 때에는 배열 변수의 크기보다 작은 수의 결과 로우를 반환하더라도 NOT FOUND 에러를 반환한다. 즉, NOT FOUND 에러를 반환하더라도 출력 배열 변수에는 결과 로우가 포함되어 있을 수 있다.

따라서 배열 변수를 사용할 때는 SQLCA의 변수인 sqlca를 사용해 루프를 빠져 나온다. 커서와 함께 배열 변수가 사용될 때에 sqlca.sqlerrd[2]에는 FETCH 문장을 수행할 때마다 현재까지 처리된 결과 로우의 누적 개수가 저장된다. 그러므로 이 누적 개수가 더 이상 증가하지 않을 때 루프를 중단하면 된다.

다음은 루프를 중단하는 소스 코드의 예이다.

[예 4.3] sqlca.sqlerrd[2]를 활용한 루프의 중단

```
int i, count;
int before_count, current_count;

...

EXEC SQL DECLARE cursor CURSOR FOR ...
...

before_count = current_count = 0;

while (1)
{
    EXEC SQL FETCH cursor INTO ...
    current_count = sqlca.sqlerrd[2];

    if (current_count == before_count) break;
    count = current_count - before_count;

    for (i = 0; i < count; i++)
    {
        /* 각 로우에 대한 처리 */
        ...
    }
    ...
    before_count = current_count;
}
```

위의 예에서는 두 개의 새로운 변수 `before_count`와 `current_count`를 사용하여 두 변수의 값이 일치하면 `while` 루프를 중단한다. 출력 배열 변수에 저장된 실제 결과 로우의 개수는 변수 `count`에 저장된다.

다수의 커서

SELECT 문장에는 배열 변수와 함께 **다수의 커서**를 사용할 수도 있다. 몇 개의 커서를 사용하더라도 하나의 커서를 사용할 때와 동일하게 처리된다. 동시에 여러 개의 커서를 사용할 때 **SQLCA**의 변수 `sqlca`가 각 커서마다 별도로 선언되지는 않는다. 따라서 변수 `sqlca`에 저장된 데이터는 직전에 실행된 질의 또는 기타 **SQL** 문장의 결과에 대한 데이터이다.

다음은 각 커서에 **FETCH**를 수행할 때마다 변수 `sqlca`에 저장되는 데이터의 예이다.

```
EXEC SQL DECLARE cursor1 CURSOR FOR ...
EXEC SQL DECLARE cursor2 CURSOR FOR ...
...
EXEC SQL OPEN cursor1;
EXEC SQL OPEN cursor2;
...
EXEC SQL FETCH cursor1 INTO :ename;
/* sqlca.sqlerrd[2] = 20 */
EXEC SQL FETCH cursor2 INTO :salary;
/* sqlca.sqlerrd[2] = 30, not 50 */
EXEC SQL FETCH cursor1 INTO :ename;
/* sqlca.sqlerrd[2] = 40, not 70 */
EXEC SQL FETCH cursor1 INTO :ename;
/* sqlca.sqlerrd[2] = 60, not 90 */
EXEC SQL FETCH cursor2 INTO :salary;
/* sqlca.sqlerrd[2] = 60, not 120 */
```

위의 예에서 배열 변수 `ename`과 `salary`의 크기는 각각 20과 30이며, `cursor1`과 `cursor2`에 연관된 **SELECT** 문장의 결과 로우의 개수가 충분히 크다고 가정한다.

스크롤 가능 커서

SELECT 문장에는 배열 변수와 함께 **스크롤 가능 커서**를 사용할 수도 있다. 스크롤 가능 커서는 일반 커서의 경우와 거의 동일하게 사용할 수 있으나, `sqlca.sqlerrd[2]`에 저장되는 값의 의미가 달라진다.

일반 커서와 스크롤 가능 커서의 `sqlca.sqlerrd[2]` 값의 차이는 다음과 같다.

구분	설명
일반 커서의 <code>sqlca.sqlerrd[2]</code>	일반 커서의 경우에는 액세스된 결과 로우의 누적 개수 를 저장하고 있다.
스크롤 가능 커서의 <code>sqlca.sqlerrd[2]</code>	현재까지 액세스된 가장 마지막 결과 로우의 절대 위치 를 저장하고 있다.

참고

일반 커서의 `sqlca.sqlerrd[2]`와 관련된 예제는 [\[예 4.2\]](#)을 참고한다. `sqlca`에 대한 자세한 내용은 “7.3.1. SQLCA 구조체 변수”를 참고한다.

스크롤 가능 커서를 사용할 때 `sqlca.sqlerrd[2]`에 저장되는 값은 다음과 같이 결정된다.

`FETCH` 문장을 수행할 때마다 옵션에 의하여 정해진 결과 로우의 위치로부터 배열 변수의 크기만큼 액세스하게 되므로, `sqlca.sqlerrd[2]`에 저장되는 값은(엑세스하고자 하는 결과 로우의 절대 위치 + 배열 변수의 크기 - 1) 값 중에서 현재까지 가장 큰 값이 된다. 만약 액세스를 하려는 위치에서부터 남아 있는 결과 로우의 개수가 배열 변수의 크기보다 작다면 `sqlca.sqlerrd[2]`에 저장되는 값은 전체 결과 로우의 개수가 된다. 결과 로우의 절대 위치 값과 `sqlca.sqlerrd[2]` 값은 항상 1 이상이다.

다음의 소스 코드는 배열 변수와 함께 스크롤 가능 커서를 사용하는 예이다.

```
EXEC SQL DECLARE cursor SCROLL CURSOR FOR ...
...
EXEC SQL OPEN cursor;
...
EXEC SQL FETCH cursor INTO :ename;
/* 1번째 로우부터 액세스. sqlca.sqlerrd[2] = 20 */

EXEC SQL FETCH NEXT cursor INTO :ename;
/* 21번째 로우부터 액세스. sqlca.sqlerrd[2] = 40 */

EXEC SQL FETCH ABSOLUTE 11 cursor INTO :ename;
/* 11번째 로우부터 액세스. sqlca.sqlerrd[2] = 40 */

EXEC SQL FETCH RELATIVE 20 cursor INTO :ename;
/* 51번째 로우부터 액세스. sqlca.sqlerrd[2] = 70 */

EXEC SQL FETCH ABSOLUTE 41 cursor INTO :ename;
/* 41번째 로우부터 액세스. sqlca.sqlerrd[2] = 70 */

EXEC SQL FETCH RELATIVE 20 cursor INTO :ename;
/* 81번째 로우부터 액세스. sqlca.sqlerrd[2] = 90 */

EXEC SQL FETCH ABSOLUTE 51 cursor INTO :ename;
/* 51번째 로우부터 액세스. sqlca.sqlerrd[2] = 90 */
```

위의 예에서 배열 변수 `ename`의 크기는 20이며, 커서에 연관된 `SELECT` 문장의 결과로 반환된 로우의 전체 개수는 90이라고 가정한다.

지시자 배열 변수

SELECT 문장에는 출력 배열 변수와 함께 **지시자 배열 변수**를 사용할 수 있다.

지시자 배열 변수는 출력 배열 변수와 같은 크기를 가져야 하며, 배열 변수로 선언한다는 것 외에는 일반적인 지시자 변수와 동일하게 사용할 수 있다. 반환되는 컬럼 값이 NULL이거나 값의 일부가 잘릴 가능성이 있다면 지시자 배열 변수의 사용을 고려해야 한다. 지시자 배열 변수도 다른 변수와 마찬가지로 **DECLARE** 영역 내에 선언한다.

다음은 지시자 배열 변수를 사용한 예이다.

```
VARCHAR addr[50][32];
short addr_ind[50];
...
EXEC SQL SELECT ADDR, ...
        INTO :addr INDICATOR :addr_ind, ...
        FROM EMP
        WHERE DEPTNO = 5;
```

참고

입력 배열 변수와 함께 지시자 배열 변수를 사용할 수도 있다. 입력 지시자 변수는 **INSERT**, **UPDATE**, **DELETE** 문장에서 사용할 수 있다.

사용 예제

다음은 [\[예 3.14\]](#)에서 제시한 예제를 배열 변수를 이용하도록 수정한 예이다.

```
#include <stdio.h>
#include <sqlca.h>
#include <string.h>
#define USERPASS    "tibero/tmax"
int main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userpass[20] = { strlen(USERPASS), USERPASS };
    int deptno;
    VARCHAR ename[30][24];
    int salary[30];
    VARCHAR addr[30][32];
    EXEC SQL END DECLARE SECTION;

    int i, count;
    int before_count, current_count;

    EXEC SQL DECLARE emp_cursor CURSOR FOR
```

```

        SELECT ENAME, SALARY, ADDR
        FROM EMP
        WHERE DEPTNO = :deptno;
EXEC SQL CONNECT :userpass;
printf("Connected.\n");
printf("Enter dept number to show: ");
scanf("%d", &deptno);

EXEC SQL OPEN emp_cursor;
before_count = current_count = 0;

while (1)
{
    EXEC SQL FETCH emp_cursor
        INTO :ename, :salary, :addr;

    current_count = sqlca.sqlerrd[2];
    if (current_count == before_count) break;          ... ② ...

    count = current_count - before_count;
    for (i = 0; i < count; i++)
    {
        printf("ename = %s, salary = %d, addr = %s\n",    ... ③ ...
            ename[i].arr, salary[i], addr[i].arr);
    }
    before_count = current_count;
}
EXEC SQL CLOSE emp_cursor;
EXEC SQL COMMIT WORK RELEASE;
}

```

① [예 3.14]과 비교했을 때 변수를 배열 변수로 선언한다.

② FETCH를 수행할 때 루프를 중단한다.

③ 배열 변수에 저장된 컬럼의 값을 printf 문을 통해 출력한다.

4.3.2. INSERT

INSERT 문장에서 배열 변수를 사용하는 방법은 일반적인 입력 변수를 사용하는 방법과 동일하다. 입력 배열 변수가 사용되는 각 문장은 문장이 실행되기 직전에 동적으로 배열 변수에 저장된 값을 읽어 들인다. 따라서 각 문장이 실행되기 전에 반드시 배열 변수에 적절한 값을 넣어 주는 코드가 있어야 한다.

INSERT 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- INSERT 문장에서 사용되는 입력 변수는 배열 변수와 일반 변수가 함께 올 수 없다. 따라서 모든 입력 변수가 배열 변수이거나 일반 변수로만 구성되어야 한다.
- SELECT 문장에서와 마찬가지로 포인터 배열 변수를 입력 변수로 사용할 수 없다.
- SELECT 문장과 유사하게 SQLCA의 변수 sqlca.sqlerrd[2]에는 삽입된 로우의 개수가 저장되어 있다.
sqlca.sqlerrd[2]에는 항상 하나의 INSERT 문장에 의해 삽입된 로우의 개수만 저장된다.
- FOR 절을 사용해 입력 배열 변수의 크기보다 적은 개수의 로우를 삽입하도록 할 수도 있다.
예를 들면 크기가 50인 입력 배열 변수의 내용 중에서 30개의 로우 값만을 이용하는 경우이다.
- 입력 배열 변수와 함께 지시자 배열 변수를 사용할 수도 있다. 삽입해야 할 컬럼 값 중 일부만 NULL 값인 경우에는 반드시 지시자 배열 변수를 사용해야 한다.

다음은 배열 변수와 함께 INSERT 문장을 실행하는 예이다.

[예 4.4] INSERT 문장의 배열 변수

```

VARCHAR ename[50][24];
int salary[50];
...
/*입력 배열 변수의 값을 적절히 설정한다.*/
strcpy((char *) ename[0].arr, "Peter");
ename[0].len = strlen(ename[0].arr);
salary[0] = 5000;
...
EXEC SQL INSERT INTO EMP (ENAME, SALARY)
      VALUES (:ename, :salary);

```

위의 [예 4.4]에서의 INSERT 문장은 다음의 for 루프를 이용한 소스 코드와 같은 결과를 갖는다. 하지만 배열 변수를 이용하는 편이 서버와의 통신 비용을 크게 줄일 수 있으므로 성능 면에서 훨씬 효율적이다.

```

for (i = 0; i < 50; i++){
    EXEC SQL INSERT INTO EMP (ENAME, SALARY)
      VALUES (:ename[i], :salary[i]);
}

```

다음은 컬럼 SALARY에 대해 지시자 배열 변수를 사용한 예이다.

```

VARCHAR ename[50][24];
int salary[50];
short salary_ind[50];
...
EXEC SQL INSERT INTO EMP (ENAME, SALARY)
      VALUES (:ename, :salary INDICATOR :salary_ind);

```

4.3.3. UPDATE

UPDATE 문장에서 배열 변수를 사용하는 방법도 일반 변수를 사용하는 방법과 유사하다.

UPDATE 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- UPDATE 문장에서는 SELECT 문장과는 다르게 **WHERE 절**에도 배열 변수를 사용할 수 있다.

WHERE 절에 배열 변수를 사용하면 SET 절에도 배열 변수를 사용해야 한다. 만약 WHERE 절에 일반 변수를 사용하였다면 SET 절에도 일반 변수를 사용해야 한다.

- INSERT 문장과 유사하게, SQLCA의 변수 sqlca.sqlerrd[2]에는 갱신된 로우의 개수가 저장된다.

sqlca.sqlerrd[2]에는 항상 하나의 UPDATE 문장에 의해 갱신된 로우의 개수만 저장되며, 무결성 제약조건을 만족하기 위해 연속적으로 갱신되거나 삭제된 로우의 개수는 포함되지 않는다.

- UPDATE 문장에서 SET 절 내에 사용된 입력 변수에는 배열 변수와 일반 변수가 함께 올 수 없다. 또한 입력 변수로 포인터 배열 변수를 사용할 수 없다.
- SELECT FOR UPDATE 문장 내에서 **UPDATE ... CURRENT OF 절**과 함께 배열 변수를 사용할 수 없다.

다음은 UPDATE 문장의 SET 절과 WHERE 절에 배열 변수를 사용한 예이다.

[예 4.5] UPDATE 문장의 배열 변수

```
int salary[50];
int empno[50];
int i;
...
/*입력 배열 변수의 값을 적절히 설정한다.*/
for (i = 1; i < 50; i++)
    empno[i] = i;
...
EXEC SQL UPDATE EMP SET SALARY = :salary
        WHERE EMPNO = :empno;
```

위의 [예 4.5]은 다음과 같은 for 루프를 이용한 연속된 UPDATE 문장과 동일한 결과를 갖는다. 하지만 배열 변수를 이용하는 편이 성능 면에서 더욱 효율적이다.

```
for (i = 0; i < 50; i++)
{
    EXEC SQL UPDATE EMP SET SALARY = :salary[i]
        WHERE EMPNO = :empno[i];
}
```

4.3.4. DELETE

DELETE 문장에서 배열 변수를 사용하는 방법도 일반 변수를 사용하는 방법과 유사하다.

DELETE 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- DELETE 문장에서도 UPDATE 문장과 같이 **WHERE 절**에 배열 변수를 사용할 수 있다.
- INSERT, UPDATE 문장에서와 유사하게, `sqlca.sqlerrd[2]`에는 삭제된 로우의 개수가 저장된다.
`sqlca.sqlerrd[2]`에는 항상 하나의 DELETE 문장에 의하여 삭제된 로우의 개수만 저장되며, 무결성 제약 조건을 만족하기 위하여 연속적으로 갱신되거나 삭제된 로우의 개수는 포함되지 않는다.
- DELETE 문장의 WHERE 절에 사용될 입력 변수로 배열 변수와 일반 변수가 함께 올 수 없다.
- 입력 변수로 포인터 배열 변수를 사용할 수 없다.
- DELETE 문장에 배열 변수를 사용할 때는 **DELETE ... CURRENT OF 절**을 사용할 수 없다.

다음은 DELETE 문장의 WHERE 절에 배열 변수를 사용한 예이다.

[예 4.6] DELETE 문장의 배열 변수

```
int empno[50];
...
/*입력 배열 변수의 값을 적절히 설정한다.*/
empno[0] = 11;
empno[1] = 15;
...
EXEC SQL DELETE EMP WHERE EMPNO = :empno;
```

위의 [예 4.6]는 다음의 for 루프와 동일한 결과를 갖는다. 하지만 INSERT, UPDATE 문장과 마찬가지로 배열 변수를 이용하는 편이 성능 면에서 효율적이다.

```
for (i = 0; i < 50; i ++){
    EXEC SQL DELETE EMP
        WHERE EMPNO = :empno[i];
}
```

4.3.5. FOR 절

INSERT, DELETE, UPDATE 문장에서 입력 배열 변수를 사용할 때 배열 변수의 크기보다 적은 개수의 로우를 처리하려는 경우에 **FOR 절**을 사용한다. FOR 절은 EXEC SQL 바로 다음에 오며, 처리하고자 하는 로우의 개수를 명시한다.

FOR 절에 로우의 개수를 명시할 때 다음의 사항에 유의해야 한다.

- FOR 절에 로우의 개수를 명시할 때는 **숫자**를 명시할 수도 있으며, **변수**를 명시할 수도 있다.

변수를 명시할 경우 해당 변수는 반드시 **DECLARE 영역**에 선언되어 있어야 한다.

- FOR 절에 로우의 개수를 명시할 때 연산식을 사용해서는 안 된다.
- FOR 절의 로우 개수는 항상 입력 배열 변수의 크기보다 작아야 한다.

tbESQL/C 프로그램에서는 지정된 로우의 개수를 저장할 배열 변수의 크기를 검토하지 않는다. 만약 지정된 로우의 개수보다 배열 변수의 크기가 작다면, 내부적으로 유효하지 않은 메모리에 접근하게 되어 메모리 에러가 발생하고, 이때 tbESQL/C 프로그램이 어떠한 동작을 할지 보장할 수 없다.

다음의 소스 코드는 FOR 절을 이용하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR ename[50];
    int salary[50];
    int empno[50];
    int row_count;
EXEC SQL BEGIN DECLARE SECTION;
...
EXEC SQL FOR 20                                ... ① ...
    INSERT INTO EMP (ENAME, SALARY, ADDR)
    VALUES (:ename, :salary, NULL);
...
row_count = 30;
EXEC SQL FOR :row_count                        ... ② ...
    UPDATE EMP SET SALARY = :salary
    WHERE EMPNO = :empno;
```

① 로우의 개수를 숫자로 설정하여 FOR 절을 이용할 수 있다.

② 변수를 이용하여 로우의 개수를 설정하여 FOR 절을 이용할 수 있다. 단, 변수 row_count는 DECLARE 영역에 선언되어 있어야 한다.

다음은 FOR 절에서 로우의 개수를 명시할 때 임의의 연산식을 사용한 경우로 잘못된 예이다.

```
EXEC SQL FOR :row_count + 10
DELETE EMP WHERE EMPNO = :empno;
```

4.4. 구조체 배열 변수

tbESQL/C 문장 내에서 여러 컬럼을 동시에 처리할 때 각 컬럼별 입/출력 변수 각각을 나열할 수도 있지만, 각 컬럼에 대한 변수를 모아 하나의 구조체로 정의하여 구조체 타입의 변수를 이용할 수도 있다.

또한 더 나아가 각 컬럼의 여러 로우를 한꺼번에 처리하고자 할 때 구조체로 정의한 타입을 배열로 선언하여 **구조체 배열 변수**를 이용할 수 있다.

4.4.1. 구조체 배열 변수의 선언

구조체 배열 변수를 선언할 때는 다음의 사항에 유의해야 한다.

- 반드시 구조체 **태그(tag)**를 삽입해야 한다.
- 구조체 배열 변수는 반드시 **DECLARE 영역**에 선언되어야 한다.
- tbESQL/C 문장에서 사용되는 구조체 타입의 변수는 중첩되어 정의될 수 없다(일반적인 구조체 변수일 때나 구조체 배열 변수일 때나 동일하다).

다음은 구조체 태그를 삽입한 예이다.

[예 4.7] 구조체 태그의 삽입

```
struct tag_emp
{
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[36];
} emp[50];
```

위의 예에서는 struct 예약어 다음에 tag_emp라는 태그를 삽입했다. 만약 구조체 태그를 넣지 않으면 프리컴파일 과정에서 에러가 발생한다.

다음은 구조체를 잘못 선언한 예이다.

```
struct tag_emsp
{
    VARCHAR ename[24];
    struct
    {
        int salary;
        VARCHAR addr[36];
    } sub_info;
} emp[50];
```

위의 예에서는 구조체 emp 내부에 구조체 sub_info를 중첩하여 정의하였다. 구조체 타입의 변수는 중첩되어 정의될 수 없다.

지시자 구조체 배열 변수를 선언할 수도 있다. 지시자 구조체 배열 변수를 선언할 때도 구조체 배열 변수를 선언할 때의 유의점을 따라야 한다. 그 밖에는 지시자 구조체 타입의 변수를 선언하는 방법과 유사하다. 추가적으로 지시자 구조체 배열 변수를 선언할 때는 다음의 사항에 유의해야 한다.

- 지시자 구조체 타입의 변수 내의 멤버 변수의 개수는 입/출력 구조체 타입의 변수 내의 멤버 변수의 개수와 같아야 하며, 각 멤버 변수는 서로 순서에 맞게 대응되도록 정의해야 한다.
- 지시자 변수는 항상 **short 타입**을 가지며, 지시자 구조체 배열 변수도 반드시 **DECLARE 영역** 내에 선언되어야 한다.

다음은 위의 [예 4.7]에서 선언한 emp 구조체 배열 변수를 지시자 배열 변수 형태로 선언한 예이다.

```
struct tag_emp_ind
{
    short ename_ind;
    short salary_ind;
    short addr_ind;
} emp_ind[50];
```

4.4.2. 사용 방법

단순 구조체 타입의 변수와 마찬가지로 구조체 배열 변수는 SELECT 문장에서의 출력 변수와 INSERT 문장에서의 입력 변수로 사용할 수 있다. 두 가지 경우 모두 지시자 구조체 배열 변수를 함께 사용할 수 있다. 입/출력 값 중에 NULL이 포함되거나 출력 값 중에 일부가 잘릴 가능성이 있다면 반드시 지시자 구조체 배열 변수를 사용해야 한다.

구조체 배열 변수를 사용하는 방법은 일반적인 배열 변수와 동일하며, 배열 변수의 사용 방법에 대한 내용은 구조체 배열 변수에도 동일하게 해당된다. 단순히 구조체 배열 변수만을 사용할 수도 있으며 커서와 함께 사용할 수도 있다.

다음은 구조체 배열 변수를 입/출력 변수로 사용하는 예이다.

```
EXEC SQL SELECT ENAME, SALARY, ADDR
          INTO :emp
          FROM EMP
          WHERE SALARY >= 50000;
...
EXEC SQL INSERT INTO EMP
          VALUES (:emp INDICATOR :emp_ind);
```

위에서는 INSERT 문장에 입력 배열 변수 emp와 지시자 구조체 배열 변수 emp_ind를 함께 사용하였다.

구조체 배열 변수 대신에 구조체 배열 변수에 대한 포인터 변수를 사용할 수도 있다. 단, 포인터의 배열 변수는 사용할 수 없다. 이러한 포인터 변수는 구조체 배열 변수를 함수 파라미터로 넘길 때에 유용하다.

다음은 포인터 변수를 사용하는 예이다.

```
struct tag_emp
{
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[36];
} emp[50], *emp_ptr;

int for_size;
...

emp_ptr = &emp[0];
for_size = sizeof(emp) / sizeof(emp[0]);

EXEC SQL FOR :for_size
    SELECT ENAME, SALARY, ADDR
    INTO :emp_ptr;
    FROM EMP
    WHERE SALARY >= 50000;
```

제5장 멀티 스레드 프로그래밍

본 장에서는 tbESQL/C 환경에서 멀티 스레드 프로그램을 작성하는 방법을 설명한다.

5.1. 개요

tbESQL/C를 이용하여 **멀티 스레드(Multi-Thread) 프로그램**을 작성할 수 있다. 멀티 스레드 프로그램에서는 스레드가 공유하는 리소스에 대해 **상호 배제(Mutual Exclusion)**를 지키기 위한 **동기화(Synchronization)** 기법이 중요하다.

참고

상호 배제란 여러 개의 스레드가 공통의 리소스에 접근할 때 리소스가 잘못된 조작으로 훼손되는 것을 방지하기 위해, 한 번에 하나의 스레드만 리소스에 접근하도록 제어하는 것을 말한다.

만약 하나의 **런타임 컨텍스트(Runtime Context)**를 동시에 여러 스레드가 접근할 경우 사용자가 **뮤텍스(Mutex)** 등을 이용하여 동시 접근을 막아야 한다.

뮤텍스는 공통의 리소스에 동시에 여러 스레드가 접근하는 것이 아닌, 한 스레드만 접근할 필요가 있을 때 사용한다. 해당 리소스에 대한 뮤텍스 객체를 생성하고, 뮤텍스 객체가 비신호 상태이면 현재 스레드가 리소스를 사용한 후 이를 반환하고, 뮤텍스 객체가 신호 상태이면, 현재 스레드를 대기 상태로 만든다.

tbESQL/C에서는 런타임 컨텍스트를 각 스레드마다 할당해 스레드 간에 상호 배제를 보장할 수 있도록 한다.

5.1.1. 런타임 컨텍스트

tbESQL/C에서는 실행 중인 애플리케이션의 접속 정보와 세션(Session) 정보, 커서 정보 등을 저장하기 위하여 **런타임 컨텍스트**를 제공한다.

싱글 스레드(Single-Thread) 기반의 tbESQL/C 프로그램에서는 사용자가 할당하지 않아도 디폴트로 설정된 런타임 컨텍스트가 사용된다. 멀티 스레드 프로그램의 경우 직접 런타임 컨텍스트를 할당하여 각 스레드마다 자신이 사용할 런타임 컨텍스트를 선택할 수 있고, 스레드 간에 런타임 컨텍스트 정보의 전달이 가능하다.

사용자는 하나의 런타임 컨텍스트를 공유하는 멀티 스레드 프로그램을 작성할 수 있으나, 이 경우 런타임 컨텍스트에 접근할 때 뮤텍스 등을 이용하여 동시 접근을 막아야 한다. 그렇게 하지 않으려면 각 스레드마다 런타임 컨텍스트를 할당해 주면 된다.

5.2. 프리컴파일러 옵션과 tbESQL/C 문장

tbESQL/C에서는 멀티 스레드와 관련하여 **프리컴파일러 옵션**과 **tbESQL/C 문장**을 제공한다.

5.2.1. 프리컴파일러 옵션

스레드와 관련된 프리컴파일러 옵션에는 THREADS가 있다.

멀티 스레드 프로그램을 작성하기 위해서는 프리컴파일할 때 THREADS 옵션을 **'YES'**로 설정해야 한다. 만약 THREADS 옵션을 **'NO'**로 설정하고, 멀티 스레드와 관련된 tbESQL/C 문장을 가진 소스 코드를 프리컴파일할 경우 에러가 발생한다.

또한 tbESQL/C가 사용하는 tbCLI 라이브러리의 경우 THREADS 옵션을 YES로 설정해야 컨텍스트 관련 문장을 사용할 수 있으며 스레드 사용과 관련된 리소스가 안전한 상태로 라이브러리를 수행한다.

5.2.2. tbESQL/C 문장

스레드와 관련된 tbESQL/C 문장을 설명하기 전에, **ctx_var**에 대해 설명하면, ctx_var는 tbESQL에서 제공하는 **sql_context 타입**의 호스트 변수이다. 이 변수는 해당 런타임 컨텍스트를 가리킨다.

ctx_var는 다음과 같이 선언한다.

```
sql_context ctx_var;
```

스레드와 관련된 tbESQL/C 문장은 다음과 같다.

ENABLE THREADS

ENABLE THREADS 문장이 명시된 후부터 실행되는 tbESQL/C 문장은 멀티 스레드 환경에서 실행된다는 것을 의미한다. 스레드와 관련된 모든 tbESQL/C 문장은 이 문장이 명시된 후에 사용할 수 있다.

다음은 ENABLE THREADS 문장을 사용하는 예이다.

```
EXEC SQL ENABLE THREADS;
```

CONTEXT ALLOCATE

sql_context 타입의 ctx_var 호스트 변수에 런타임 컨텍스트를 할당한다. 할당된 ctx_var는 CONTEXT USE, CONTEXT FREE에 사용될 수 있다.

다음은 CONTEXT ALLOCATE 문장을 사용하는 예이다.

```
EXEC SQL CONTEXT ALLOCATE :ctx_var;
```

CONTEXT USE

EXEC SQL CONTEXT USE 문장을 통해 이 문장 이후의 ESQL 문장에서 사용할 런타임 컨텍스트를 지정해 줄 수 있다. 이 문장의 적용 범위는 EXEC SQL WHENEVER 문장과 마찬가지로 다음에 나올 EXEC SQL CONTEXT USE 문장 직전까지이다.

ctx_var는 EXEC SQL CONTEXT ALLOCATE를 통해 런타임 컨텍스트를 할당 받은 변수이어야 하고, 'DEFAULT'로 설정할 경우 디폴트로 설정된 런타임 컨텍스트를 사용하겠다는 의미이다. 모든 tbESQL/C 애플리케이션은 실행될 때 디폴트 런타임 컨텍스트를 할당 받는다.

다음은 CONTEXT USE 문장을 사용하는 예이다.

```
EXEC SQL CONTEXT USE { :ctx_var | DEFAULT };
```

CONTEXT FREE

sql_context 타입의 ctx_var 호스트 변수에 할당된 런타임 컨텍스트를 해제한다. ctx_var는 EXEC SQL CONTEXT ALLOCATE를 통해 런타임 컨텍스트가 할당된 변수여야 한다.

다음은 CONTEXT FREE 문장을 사용하는 예이다.

```
EXEC SQL CONTEXT FREE :ctx_var;
```

5.3. 주의사항

멀티 스레드 프로그램을 작성할 경우 각각의 스레드가 공유하는 리소스에 대한 **상호 배제**를 보장하는 것이 중요하다.

sqlca, sqllda와 같이 전역(global)으로 선언된 변수의 경우 tbESQL/C 런타임 라이브러리(Runtime Library)에서는 상호 배제를 보장하지 않는다. 그러므로 각 런타임 컨텍스트가 자기만의 변수를 처리할 수 있도록 다음과 같이 프로그램을 작성해야 한다.

```
sql_context ctx;
...
void do_update()
{
    struct sqlca sqlca;
    /* 지역(local) 변수로 선언하여 현재 스레드(런타임 컨텍스트)만의
       변수로 사용한다. */
    ...
    EXEC SQL CONTEXT USE :ctx;
    EXEC SQL UPDATE TEST SET COL1 = 'thread'
        WHERE COL2 = 2.5;
    if (sqlca.sqlcode < 0) do_error();
}
...
```

5.4. 사용 예제

다음은 스레드 관련된 프리컴파일러 옵션과 tbESQL/C 문장을 사용하는 예이다.

```
sql_context ctx1;
sql_context ctx2;

EXEC SQL ENABLE THREADS;
EXEC SQL CONTEXT ALLOCATE :ctx1;
EXEC SQL CONTEXT ALLOCATE :ctx2;

EXEC SQL INSERT INTO TEST VALUES('test1');
/* 디폴트 런타임 컨텍스트에 적용된다. */

EXEC SQL CONTEXT USE :ctx1;
EXEC SQL INSERT INTO TEST VALUES('test2');
/* ctx1 런타임 컨텍스트에 적용된다. */

EXEC SQL CONTEXT USE :ctx2;
EXEC SQL INSERT INTO TEST VALUES('test3');
/* ctx2 런타임 컨텍스트에 적용된다. */

EXEC SQL COMMIT;
/* ctx2 런타임 컨텍스트를 커밋한다. */

EXEC SQL CONTEXT FREE :ctx1;
EXEC SQL CONTEXT FREE :ctx2;
```

제6장 Dynamic SQL

본 장에서는 프로그램이 실행될 때 SQL 문장의 내용을 동적으로 지정하는 Dynamic SQL을 설명한다.

6.1. 개요

tbESQL/C 프로그램은 컴파일을 할 때 미리 정해진 완전한 SQL 문장을 실행할 수도 있지만 일부 애플리케이션에서는 컴파일을 할 때가 아닌 프로그램을 실행할 때 완전한 SQL 문장이 정해지는 경우도 있다.

Dynamic SQL 문장은 이러한 경우를 위해 프로그램을 실행할 때 사용자의 입력 등에 따라 유연하게 SQL 문장을 완성하여 실행하는 인터페이스이다.

Dynamic SQL은 SELECT 문장인지 아닌지, 또 문장 내에 입력 변수의 포함 여부에 따라 다른 순서로 실행된다. 이때 입력 변수와 출력 변수의 개수와 데이터 타입은 컴파일을 할 때 알고 있어야 한다. Dynamic SQL 문장은 프로그램을 실행할 때 완성되므로 WHERE 절 등의 조건식을 동적으로 변경할 수도 있다.

Dynamic SQL을 사용하는 경우는 다음의 두 가지이다.

- SQL 문장의 일부가 완성되지 않은 경우

예를 들어 UPDATE 문장의 WHERE 절의 조건이 프로그램이 실행될 때 정해지는 경우를 들 수 있다.

- SQL 문장을 실행할 대상이 정해지지 않은 경우

예를 들어 GRANT 문장을 이용하여 특권을 부여할 사용자가 프로그램이 실행될 때 정해지는 경우를 들 수 있다.

6.2. 특징

Dynamic SQL 문장은 다음과 같은 특징이 있다.

- 문자열 데이터이다.

tbESQL/C 프로그램에서 EXEC SQL을 통해 실행되는 일반 SQL 문장은 프로그램 내에 직접 포함되지 만, Dynamic SQL 문장은 문자열 타입 데이터 형태로 포함된다. 즉, 문자열 변수에 저장되어 있거나 문자열 값으로 표현된다.

- 시작과 끝을 나타내는 부호가 없다.

Dynamic SQL 문장 내에는 시작을 나타내는 EXEC SQL이나 끝을 나타내는 세미콜론(;)이 포함되지 않는다.

다음은 Dynamic SQL 문장의 몇 가지 예이다.

```
'DELETE FROM EMP WHERE SALARY < 20000'  
'UPDATE EMP SET SALARY = SALARY * 1.05 WHERE DEPTNO = :deptno'  
'SELECT ENAME, ADDR, SALARY FROM EMP WHERE DEPTNO = :deptno'
```

6.3. 실행 방법

Dynamic SQL 문장을 실행하는 방법에는 다음의 4가지가 있다.

- **방법 1**

입력 변수를 포함하지 않는 SELECT 이외의 문장을 실행한다.

- **방법 2**

입력 변수를 포함하는 SELECT 이외의 문장을 실행한다.

- **방법 3**

SELECT 문장의 실행 방법이며, 입력 변수를 포함할 수도 있고 포함하지 않을 수도 있다.

- **방법 4**

SELECT 문장으로 조회할 컬럼의 개수 및 데이터 타입을 프리컴파일 시점에서 알 수 없을 때 사용한다.

6.3.1. 방법 1

방법 1은 입력 변수를 포함하지 않는 SELECT 이외의 문장을 동적으로 실행하는 방법이다. 이 방법은 하나의 SQL 문장을 실행할 때 동적으로 생성하여 EXECUTE IMMEDIATE 문장을 통하여 실행한다. 이 방법은 동일한 SQL 문장을 실행하더라도 실행할 때마다 다시 파싱(Parsing)과 최적화를 수행한다.

SQL 문장을 실행하기 위한 **EXECUTE IMMEDIATE의 문법**은 다음과 같다.

```
EXEC SQL EXECUTE IMMEDIATE { :sql_var | 'sql_stmt' }
```

항목	설명
sql_var	SQL 문장을 포함하는 문자열 타입 변수이다. VARCHAR 타입이거나 CHAR 타입 모두 가능하다.
sql_stmt	문자열 형태의 SQL 문장이다.

사용 예제

다음은 방법 1을 통해 실행 가능한 SQL 문장의 예이다.

```
'DELETE FROM EMP WHERE SALARY < 20000'
```

```
'GRANT SELECT ANY TABLE TO John'
```

다음은 방법 1을 통해 SQL 문장을 실행하는 예이다.

```
char sql_var1[128];
VARCHAR sql_var2[100];
...
strcpy(sql_var1,
        "UPDATE EMP SET SALARY = SALARY * 1.05 WHERE DEPTNO = 5");
EXEC SQL EXECUTE IMMEDIATE :sql_var1;
...
strcpy(sql_var2.arr,
        "UPDATE EMP SET SALARY = SALARY * 1.05 WHERE DEPTNO = 6");
sql_var2.len = strlen(sql_var2.arr);
EXEC SQL EXECUTE IMMEDIATE :sql_var2;
...
EXEC SQL EXECUTE IMMEDIATE 'GRANT SELECT ANY TABLE TO John';
```

VARCHAR 타입의 변수인 경우에는 len 변수의 값도 정확하게 설정하여야 한다. 위의 예에서는 소스 코드 'sql_var2.len = strlen(sql_var2.arr);'을 통해 len 변수의 값을 설정하고 있다.

방법 1을 통해 SQL 문장을 실행할 때는 실행할 때마다 다시 파싱과 최적화 과정을 거치기 때문에 DDL 문장처럼 한 번만 실행하기 위한 문장에 적합하다. 만약 SQL 문장 내의 일부 값만을 변화시켜 실행하려면, 다음 절에서 설명할 방법 2가 더 적합하다.

6.3.2. 방법 2

방법 2는 입력 변수를 포함하는 SELECT 이외의 문장을 동적으로 실행하는 방법이다.

이 방법을 사용하면, 입력 변수만 다르고 나머지 내용은 동일한 SQL 문장을 실행하는데 유용하다. 그 이유는 문장마다 매번 파싱을 수행하지 않고, 파싱을 한 번만 수행한 뒤 바로 최적화를 수행하기 때문이다.

이 방법을 통하여 실행하는 SQL 문장 내의 입력 변수의 개수와 데이터 타입에는 제한이 없으나, 컴파일을 할 때 미리 알고 있어야 한다. 입력 변수가 없는 SELECT 문장 이외의 다른 문장을 방법 2를 통하여 실행할 수도 있다. 하지만 PREPARE와 EXECUTE의 두 단계를 거쳐야 하므로, 그러한 문장은 방법 1을 통해 실행하는 것이 유리하다.

이 방법은 Dynamic SQL 문장을 다음의 두 단계에 걸쳐 실행한다.

1. PREPARE 단계

SQL 문장을 준비하기 위한 PREPARE의 문법은 다음과 같다.

```
EXEC SQL PREPARE stmt_name FROM { :sql_var | 'sql_stmt' }
```

항목	설명
stmt_name	준비된 SQL 문장을 대표하는 이름이며 EXECUTE 문장에서 사용된다.
sql_var	SQL 문장을 포함하는 문자열 타입 변수이다. VARCHAR 타입이거나 CHAR 타입 모두 가능하다.
sql_stmt	문자열 형태의 SQL 문장이다.

PREPARE 문을 통하여 준비된 SQL 문장은 COMMIT 또는 ROLLBACK 문을 실행한 후에도 사용할 수 있다. 이러한 문장들은 현재 세션이 끝날 때까지 유효하다.

2. EXECUTE 단계

준비된 SQL 문장을 실행하기 위한 EXECUTE의 문법은 다음과 같다.

```
EXEC SQL EXECUTE stmt_name [USING :host_var[:ind_var], ...]
```

항목	설명
stmt_name	실행할 SQL 문장이며, PREPARE를 사용해 준비된 문장의 이름이다.
USING	실행하려는 SQL 문장에 입력 변수가 있는 경우 입력 변수의 값을 할당하기 위해 사용된다. USING 절에 나열되는 변수의 개수와 데이터 타입은 SQL 문장 내의 입력 변수의 개수와 데이터 타입과 동일해야 한다. USING 절에 나열된 변수의 데이터 타입이 준비된 SQL 문장의 입력 변수의 데이터 타입과 다르더라도 데이터 값의 변환이 가능하다면 해당 변수의 사용이 가능하다. 예를 들어 USING에 나열된 변수의 데이터 타입이 NUMBER이며, 이에 대응되는 입력 변수의 데이터 타입이 VARCHAR라면, 데이터 값이 변환되어 입력 변수에 할당된다. USING 절에서 변수를 나열할 때 지시자 변수를 함께 포함할 수도 있다. 지시자 변수는 NULL 값 등을 표현할 경우에 사용될 수 있다. USING 절의 변수는 배열 변수를 사용할 수도 있다. 이때 USING 절 내의 모든 변수가 배열 변수이어야 하며 같은 크기를 가져야 한다.
host_var	입력 변수로 사용될 호스트 변수이다.
ind_var	지시자 변수이다.

문장 이름 **stmt_name**은 PREPARE를 이용해 여러 SQL 문장을 준비한 경우에 EXECUTE 문장에서 하나의 문장을 지정하기 위하여 사용한다. 즉, 문장 이름은 프로그램 모듈 내에서 유일하게 선언해야 한다. 문장 이름은 프로그램 변수가 아니며 DECLARE 영역 내에서 선언하지 않고, PREPARE 문장 내에서만 선언된다.

참고

1. **tbESQL/C** 프로그램에서의 데이터 타입 변환은 “[2.3. tbESQL/C 데이터 타입](#)”을 참고한다.
2. 지시자 변수에 대해서는 “[2.3.8. 지시자](#)”를 참고한다.

사용 예제

다음은 방법 2를 통해 실행 가능한 SQL 문장의 예이다.

```
'UPDATE EMP SET SALARY = SALARY * 1.05 WHERE DEPTNO = :deptno'
```

```
'INSERT INTO EMP(ENAME, DEPTNO) VALUES (:ename, :deptno)'
```

다음은 방법 2를 통해 SQL 문장을 실행하는 예이다.

```
char sql_var[128];
double sal_ratio; int deptno;
...
strcpy(sql_var, "UPDATE EMP SET SALARY = SALARY * :sal_ratio"
        "WHERE DEPTNO = :deptno");
EXEC SQL PREPARE update_emp FROM :sql_var;
...

sal_ratio = 1.05; deptno = 5;
EXEC SQL EXECUTE update_emp USING :sal_ratio, :deptno;

sal_ratio = 1.08; deptno = 6;
EXEC SQL EXECUTE update_emp USING :sal_ratio, :deptno;
```

위의 예에서는 두 개의 입력 변수를 갖는 UPDATE 문장 하나를 두 번 실행하였다.

6.3.3. 방법 3

방법 3은 SELECT 문장을 동적으로 실행하는 방법이다. 입력 변수를 포함할 수도 있고 포함하지 않을 수도 있다. 동적 SELECT 문장에 포함되는 입/출력 변수의 개수와 데이터 타입에는 제한이 없으나, 컴파일을 할 때 미리 알고 있어야 한다. 하지만 FROM 절에 포함되는 테이블과 뷰, WHERE 절 내의 조건식, GROUP BY 절, ORDER BY 절 등은 실행을 할 때 정해져도 무방하다.

방법 3은 SELECT 문의 실행 결과를 얻기 위하여 **커서**를 사용하기 때문에, Dynamic SQL 문장을 실행할 때 다섯 단계를 거친다. 전체적인 실행 순서를 설명하면 다음과 같다.

1. PREPARE 단계

PREPARE를 실행한다. PREPARE는 주어진 SQL 문장을 해석하고 실행할 준비를 한다. PREPARE의 문법 및 사용 방법은 “6.3.2. 방법 2”와 동일하다.

2. DECLARE 단계

SQL 문장을 준비한 후에는 DECLARE를 이용하여 커서 또는 스크롤 가능 커서를 선언한다.

DECLARE의 문법은 다음과 같다.

```
DECLARE cursor_name [SCROLL] CURSOR FOR stmt_name
```

항목	설명
cursor_name	선언하는 커서의 이름이다. 커서 이름은 변수가 아니며 DECLARE 내에서만 선언된다. DECLARE를 이용해 여러 개의 커서를 선언한 경우에 OPEN, FETCH, CLOSE 등의 문장에서 하나의 커서를 지정하기 위하여 사용한다.
stmt_name	PREPARE를 통해 준비된 SELECT 문장의 이름이다.

3. OPEN 단계

커서를 선언한 후에 커서에 연관된 SELECT 문장을 실행하기 위하여 **OPEN**을 실행한다. OPEN은 먼저 USING 절에 나열된 변수 값을 SELECT 문장의 입력 변수에 할당하고, 커서에 메모리 영역을 할당한 뒤 SELECT 문장을 실행한다. OPEN 문은 커서를 질의 결과 로우의 맨 처음 로우에 위치시킨다.

OPEN의 문법은 다음과 같다.

```
OPEN cursor_name [USING :host_var[:ind_var], ...]
```

항목	설명
cursor_name	OPEN을 통해서 열 커서의 이름이다.
USING	SELECT 문장의 입력 변수로 사용될 변수를 나열할 때 사용한다.
host_var	입력 변수로 사용될 호스트 변수이다.
ind_var	지시자 변수이다.

커서 이름은 SQL 문장의 이름과 연관되지만, 문장 이름이 의미하는 실제 SQL 문장과는 직접적으로 연관되어 있지 않다. 따라서, 어떤 커서를 문장 이름에 대해 선언한 후 그 문장 이름이 의미하는 실제 SQL 문장이 변경되었을 경우에도 커서에 OPEN을 수행하면 변경된 SQL 문장이 실행된다.

다음은 이러한 커서의 이름과 문장의 이름 사이의 연관에 대한 예이다.

```
EXEC SQL PREPARE stmt1 FROM :sql_var1;
EXEC SQL DECLARE cursor1 FOR stmt1;
EXEC SQL PREPARE stmt1 FROM :sql_var2;
EXEC SQL OPEN cursor1;
```

위의 예의 마지막 라인에서 cursor1에 대해 OPEN을 수행하면 sql_var1이 아닌 sql_var2에 저장된 SQL 문장이 실행된다.

4. FETCH 단계

질의 결과 로우를 하나씩 추출하기 위하여 FETCH를 이용한다. 추출된 로우의 각 컬럼 값은 INTO 절 내의 변수에 저장된다. 지시자 변수는 NULL 값 등을 표현하기 위하여 선택적으로 사용된다. 만약 질의 결과 로우가 하나도 없거나 더 이상 읽을 로우가 없다면 NOT FOUND 에러를 반환한다. 스크롤 가능 커서로 선언한 경우에는 원하는 특정 위치의 로우를 선택하여 읽을 수 있다.

FETCH의 문법은 다음과 같다.

```
FETCH [ FIRST | LAST | PRIOR | NEXT | CURRENT | RELATIVE offset
      | ABSOLUTE offset ] cursor_name INTO :host_var[:ind_var], ...
```

커서 이름 바로 앞에 있는 옵션은 액세스하려는 로우의 위치를 의미한다. 스크롤 가능 커서에 대한 설명과 각 옵션의 의미는 “3.5. 스크롤 가능 커서”를 참고한다.

5. CLOSE 단계

커서를 이용해 원하는 모든 로우를 읽은 후에는 해당 커서에 CLOSE를 실행한다. CLOSE는 커서에 할당된 메모리를 반환하고 커서를 닫는다. 닫힌 커서에 대해서는 FETCH를 실행할 수 없다.

CLOSE의 문법은 다음과 같다.

```
CLOSE cursor_name
```

항목	설명
cursor_name	CLOSE를 통해서 닫을 커서의 이름이다.

사용 예제

다음은 방법 3을 통해 실행 가능한 SQL 문장의 예이다.

```
'SELECT ENAME, ADDR, SALARY FROM EMP WHERE DEPTNO = 5'
```

```
'SELECT DNAME, LOC FROM DEPT WHERE DEPTNO = :deptno'
```

다음은 방법 3을 통해 SQL 문장을 실행하는 예이다.

```
char sql_var[128];
int deptno;
VARCHAR ename[24];
double salary;
VARCHAR addr[36];

...
strcpy(sql_var, "SELECT ENAME, SALARY, ADDR"
        "FROM EMP WHERE DEPTNO = :deptno");
EXEC SQL PREPARE select_stmt FROM :sql_var;
EXEC SQL DECLARE emp_cursor CURSOR FOR select_stmt;
...
deptno = 5;
EXEC SQL OPEN emp_cursor USING :deptno;
EXEC SQL WHENEVER NOT FOUND DO break;

while (1) {
    EXEC SQL FETCH emp_cursor INTO :ename, :salary, :addr;
```

```

    ...
}

EXEC SQL CLOSE emp_cursor;

```

위의 예에서는 'emp_cursor'라는 이름으로 커서를 선언하였으며, NOT FOUND 에러가 반환될 때까지 while 루프 내의 FETCH 문을 계속 실행한다.

6.3.4. 방법 4

방법 4는 SELECT 문장을 통해 조회할 컬럼의 개수 및 타입을 프리컴파일 시점에서 알 수 없을 때 사용하는 방법이다. 이 방법에 사용되는 핵심적인 자료 구조는 **SQLDA**라는 구조체이다. **SQLDA** 타입의 서술자 (Descriptor) 변수를 이용해서 바인드 변수와 조회할 컬럼을 기술하여 Dynamic SQL을 수행한다.

다음은 **SQLDA** 구조체의 선언 방법 및 각 멤버 변수를 주석을 통해 설명한다.

```

struct SQLDA
{
    long N;        /* 서술자 내의 항목의 개수 */
    char **V;      /* 실제 데이터가 들어가 버퍼 배열 변수 */
    long *L;       /* 버퍼의 길이를 나타내는 변수 */
    short *T;      /* 버퍼의 타입을 나타내는 변수 */
    short *I;      /* 지시자 변수를 위한 공간 */
    long F;        /* DESCRIBE에 의해서 발견된 변수의 개수 */
    char **S;      /* 변수의 이름을 저장하기 위한 배열 변수 */
    short *M;      /* 변수 이름의 최대길이를 저장하는 변수 */
    short *C;      /* 변수 이름의 현재 값을 저장하는 변수 */
    char *X;       /* 지시자 변수 이름을 위한 배열 */
    short *Y;      /* 지시자 변수이름의 최대 길이 */
    short *Z;      /* 지시자 변수 이름의 현재 길이 */
}

```

Dynamic Method 4를 사용할 때의 binding 가능한 host variable type이 정해져 있다.

다음은 **SQLDA** 구조체의 타입 변수인 T에 맵핑시킬 수 있는 External 데이터 타입과 타입 코드를 설명한다.

External Type	Type Code	C type
VARCHAR2	1	char[n]
NUMBER	2	char[n] (n < 22)
INTEGER	3	int
FLOAT	4	float
STRING	5	char[n+1]

External Type	Type Code	C type
VARNUM	6	char[n] (n < 22)
DECIMAL	7	float
LONG	8	char[n]
BINARY_FLOAT	21	float
BINARY_DOUBLE	22	double
VARCHAR	9	char[n+2]
ROWID	11	char[n]
DATE	12	char[n]
VARRAW	15	char[n]
RAW	23	unsigned char[n]
LONG RAW	24	unsigned char[n]
UNSIGNED	68	unsigned int
DISPLAY	91	char[n]
LONG VARCHAR	94	char[n+4]
LONG VARRAW	95	unsigned char[n+4]
CHAR	96	char[n]
CHARF	96	char[n]
CHARZ	97	char[n+1]
CLOB	112	Cloblocator
BLOB	113	Bloblocator

방법 4의 전체적인 실행 순서를 설명하면 다음과 같다.

1. 먼저 **SQLDA** 변수를 선언한다.

다음은 SQLDA 변수를 선언하는 예이다.

```
#include <sqllda.h>
SQLDA *select_dp;
SQLDA * bind_dp;

char *id = "T002";
char *cd = "B001";

char *stmt = "select teller_name from teller where teller_id=:id and\
            branch_cd=:cd";
```

2. 각 변수에 대하여 **SQLSQLDAAlloc** 함수를 이용해서 메모리 공간을 할당한다.

다음은 SQLSQLDAAlloc 함수를 통해 메모리 공간을 할당하는 예이다.

```
bind_dp = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t)5, 4);
select_dp = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t)5, (size_t)0);
```

3. 각 서술자에 사용될 컬럼의 개수와 변수의 개수를 지정한다.

다음은 서술자에 사용될 컬럼의 개수와 변수의 개수를 지정하는 예이다.

```
select_dp->N = 3;
bind_dp->N = 3;
```

4. 호스트 변수에 미리 지정된 SQL 문장을 **PREPARE**를 사용해 준비한다.

다음은 SQL 문장을 준비하는 예이다.

```
EXEC SQL PREPARE S1 FROM :stmt;
```

5. **DECLARE**를 사용해 커서를 선언한다.

다음은 커서를 선언하는 예이다.

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
```

6. **DESCRIBE**로 SQL 문장의 바인드 변수를 바인딩한다.

다음은 바인드 변수를 바인딩을 하는 예이다.

```
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 INTO bind_dp;
```

7. DESCRIBE 문장을 통해 실제 계산된 바인드 변수의 개수를 다시 지정한다.

bind_dp->F의 값이 2임을 알고 난 후 다음과 같이 설정해 줄 수 있다.

```
bind_dp->N = bind_dp->F
```

8. 바인드 변수를 위한 메모리를 할당한다.

여기서 bind_dp->F의 값이 2가 아닌 다른 수였다면 그에 맞춘 값 할당이 필요하다. for 루프 등을 사용해 해당 값을 알맞게 할당하는 프로그램을 작성할 수도 있다.

다음은 바인드 변수를 위한 메모리 공간을 할당하는 예이다.

```
bind_dp->V[0] = id;
bind_dp->L[0] = strlen(id);
bind_dp->T[0] = 5;
bind_dp->V[1] = cd;
bind_dp->L[1] = strlen(cd);
bind_dp->T[1] = 5;
```

9. **OPEN**을 사용해 준비된 바인드 서술자로 커서를 연다.

다음은 바인드 서술자를 사용해 커서를 여는 예이다.

```
EXEC SQL OPEN C1 USING DESCRIPTOR bind_dp;
```

10 조회할 컬럼에 대한 서술자를 작성한다.

다음은 서술자를 작성하는 예이다.

```
EXEC SQL DESCRIBE SELECT LIST FOR S1 INTO select_dp;
```

11 실제 조회할 컬럼의 개수를 알게 되었으므로 이를 재설정한다.

다음은 컬럼의 개수를 재지정하는 예이다.

```
select_dp->N = select_dp->F;
```

12 조회할 각 컬럼의 길이와 데이터 타입을 재설정한다.

다음은 select_dp->F 값이 1, 즉 select list의 개수가 1이라고 가정하고 재설정하는 예이다.

```
select_dp->V[0] = malloc(select_dp->L[0] + 1);  
memset(select_dp->V[0], 0, select_dp->L[0] + 1);
```

select_dp->F의 개수에 따라 select_dp의 각 변수의 설정이 달라져야 한다.

13 서술자를 이용해서 **FETCH**를 실행한다.

다음은 FETCH를 실행하는 예이다.

```
EXEC SQL FETCH C1 USING DESCRIPTOR select_dp;
```

수행결과는 서술자의 V변수에 값이 저장된다.

14 FETCH의 실행을 완료한 후에는 할당된 메모리를 모두 해제한다.

다음은 할당된 메모리를 해제하는 예이다.

```
free(select_dp->V[0]);  
  
SQLSQLDAFree(SQL_SINGLE_RCTX, select_dp);  
SQLSQLDAFree(SQL_SINGLE_RCTX, bind_dp);
```

15 마지막으로 **CLOSE**를 사용해 커서를 닫는다.

다음은 커서를 닫는 예이다.

```
EXEC SQL CLOSE C1;
```

사용 예제

다음은 방법 4를 통해 배열변수를 이용한 SQL 문장을 실행하는 예이다.

```
char *stmt = "select teller_id, teller_name from teller where branch_cd=:cd \  
              order by teller_id";
```

```

char *cd = "B001";
int count = 2;

SQLDA *bind_dp;
SQLDA *select_dp;

EXEC SQL WHENEVER SQLERROR
    DO printf("file: %s, line: %d, error code: %d\n", __FILE__, __LINE__,
        sqlca.sqlcode);

bind_dp = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t)5, 4);
select_dp = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t)5, (size_t)0);

bind_dp->N = MAX_ITEMS;
select_dp->N = MAX_ITEMS;

/* prepare */
EXEC SQL PREPARE S1 FROM :stmt;

/* declare cursor */
EXEC SQL DECLARE C2 CURSOR FOR S1;

/* describe bind parameters */
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 INTO bind_dp;

bind_dp->N = bind_dp->F;

/* bind parameters */
bind_dp->V[0] = cd;
bind_dp->L[0] = strlen(cd) + 1;
bind_dp->T[0] = 5;

/* open cursor */
EXEC SQL OPEN C2 USING DESCRIPTOR bind_dp;

/* describe select list */
EXEC SQL DESCRIBE SELECT LIST FOR S1 INTO select_dp;

/* allocate bind col buffer */
select_dp->N = select_dp->F;
select_dp->V[0] = malloc(select_dp->L[0] * 2);
memset(select_dp->V[0], 0, select_dp->L[0] * 2);
select_dp->V[1] = malloc(select_dp->L[1] * 2);
memset(select_dp->V[1], 0, select_dp->L[1] * 2);
select_dp->T[1] = 5;

/* fetch */

```

```
EXEC SQL FOR :count FETCH C2 USING DESCRIPTOR select_dp;

EXEC SQL CLOSE C2;

/* free sqllda */
SQLSQLDAFree(SQL_SINGLE_RCTX, bind_dp);
SQLSQLDAFree(SQL_SINGLE_RCTX, select_dp);

EXEC SQL COMMIT RELEASE;
```


제7장 런타임 에러 처리

본 장에서는 tbESQL/C 프로그램을 실행할 때 발생할 수 있는 예외 상황을 처리하기 위한 방법인 런타임 에러 처리에 대해 설명한다.

7.1. 개요

tbESQL/C 프로그램에서 SQL 문장을 실행하는 중에 에러가 발생했을 때, 그에 대한 적절한 조치를 취하기 위해 다음의 3가지 인터페이스를 제공한다.

- 상태 변수
- SQLCA
- WHENEVER

tbESQL/C 프로그램 내에서 하나의 SQL 문장을 실행하면 항상 **상태 변수** 또는 **SQLCA**에 실행 결과 또는 에러 정보를 저장한다. 애플리케이션 프로그램 개발자는 SQL 문장을 실행한 직후에 상태 변수 또는 SQLCA를 검토하여 에러가 발생했을 때 해결을 위해 코드를 삽입해 주어야 한다.

상태 변수와 SQLCA의 변수는 SQL 문장이 실행될 때마다 자동으로 변경되고, 프로그램 내에서 그 내용을 검토해야 한다. 이러한 작업을 자동적으로 수행하기 위하여 **WHENEVER 문장**을 사용한다. WHENEVER 문장은 에러 또는 경고가 발생했을 때 그에 따른 처리를 위한 특정한 작업을 수행하도록 선언한다.

일반적인 tbESQL/C 프로그램에서는 WHENEVER 문장을 이용하여 에러 또는 경고가 발생했을 때 특정한 에러 처리 함수를 호출하도록 선언한다. 에러 처리 함수는 상태 변수 또는 SQLCA의 변수의 내용을 검토하여 적절히 조치한다. 에러 상황에 따라 프로그램을 계속 진행하거나 정지시킬 수 있다.

7.2. 상태 변수

상태 변수는 에러 또는 경고 코드를 가지며, 에러 또는 경고에 대한 상세한 정보가 저장된다.

7.2.1. 특징

상태 변수에는 SQLSTATE와 SQLCODE가 있는데, 주요 특징은 다음과 같다.

- 두 상태 변수는 **서로 다른 값**을 갖는다.

SQL 문장을 실행할 때마다 두 상태 변수의 값은 tbESQL/C 프로그램에 의해 자동적으로 갱신되는데, 같은 예외 상황이라고 해도 두 상태 변수는 서로 다른 값을 갖는다. SQLSTATE에는 **에러 및 경고 코드**가 저장되지만 SQLCODE에는 **에러 코드**만이 저장된다.

- SQLSTATE의 길이는 **5bytes**이다.

SQLSTATE는 6bytes로 선언하지만 맨 마지막에 포함되는 NULL 문자를 제외하면, 실제 SQLSTATE의 길이는 5bytes이다.

- SQLSTATE는 다음의 **두 가지 코드**로 구성된다.

코드	설명
클래스 코드	에러 또는 경고에 대한 큰 분류이며, 크기는 2bytes이다.
서브 클래스 코드	클래스 코드의 큰 분류 내부의 구체적인 내용을 포함하고 있으며, 크기는 3bytes이다.

클래스 코드와 서브 클래스 코드를 예를 들어 설명하면, SQLSTATE의 값이 '22012'라면, 클래스 코드 '22'는 데이터 예외 상황(Data exception)을 의미하며 서브 클래스 코드 '012'는 0으로 나눔(Divide by zero)을 의미한다.

- SQLSTATE에는 '0' ~ '9', 'A' ~ 'Z'로 이루어진 문자열이 저장된다.

SQL-92에서 정의된 클래스 코드와 서브 클래스 코드는 모두 '0' ~ '4'와 'A' ~ 'H'로 시작된다. 이외의 문자로 시작되는 클래스 코드와 서브 클래스 코드는 추가적인 에러 및 경고 코드를 정의하기 위한 영역이다.

7.2.2. 상태 변수 선언

프로그램 내에서 상태 변수를 사용하기 위해서는 먼저 상태 변수를 선언해야 한다.

두 가지 상태 변수를 선언하는 방법은 다음과 같다.

- SQLSTATE

SQLSTATE는 반드시 **크기 6의 char 배열 타입**을 가져야 하며, **DECLARE 영역** 안에 **대문자**로 선언해야 한다. 다음은 SQLSTATE를 선언하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
...
EXEC SQL END DECLARE SECTION;
```

- SQLCODE

SQLCODE는 반드시 **long 타입**을 가져야 하며, **DECLARE 영역** 안에 **대문자**로 선언해야 한다. 다음은 SQLCODE를 선언하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;
long SQLCODE = 0;
...
EXEC SQL END DECLARE SECTION;
```

7.2.3. 상태 변수 종류

상태 변수에는 다음의 두 가지가 있는데, SQL 문장을 실행하는 도중에 발생한 에러 또는 경고를 서로 다른 값으로 표현한다.

- SQLSTATE

SQLSTATE가 갖는 에러 또는 경고 코드는 SQL-92 표준에서 정의되어 있고, 추가적인 코드를 정의할 수 있도록 확장성을 지원하고 있다.

다음은 SQLSTATE의 클래스 코드를 정리한 것이다. 여기에서는 SQL-92 표준에서 정의된 클래스 코드만을 나열하였다.

클래스 코드	설명
00	성공적인 완료
01	경고

참고

SQLSTATE가 가질 수 있는 에러 및 경고 코드는 "Tibero tbCLI 안내서"를 참고한다.

- SQLCODE

SQLCODE는 SQL-92에서는 사용하지 않는다. SQLCODE에 저장되는 에러 코드는 정수 값을 가지며, 0인 경우와 양수인 경우, 음수인 경우로 나눌 수 있다. 각 SQLCODE의 값의 의미는 다음과 같다.

SQLCODE의 값	설명
0	SQLCODE 값이 0인 경우는 에러 없이 SQL 문장을 성공적으로 수행한 경우를 나타낸다.
양수	SQLCODE 값이 0보다 큰 경우는 SQL 문장의 수행을 마쳤으나 예외 상황이 발생했음을 의미한다. 예를 들어 SELECT 문장이 반환한 결과 로우가 하나도 없거나 UPDATE 문장을 실행한 결과 갱신된 로우가 하나도 없는 경우를 들 수 있다.
음수	SQLCODE 값이 0보다 작은 경우는 에러 상황을 나타낸다. 이 때에는 에러 때문에 SQL 문장의 수행을 끝마치지 못했음을 의미한다.

참고

음수 값을 갖는 SQLCODE 값은 Tibero에서 발생하는 일반적인 에러의 번호이다. 이에 대해서는 "Tibero 에러 참조 안내서"를 참고한다.

안내서에 기술된 에러 메시지 번호는 모두 양수이며, SQLCODE 값의 절댓값을 기준으로 찾는다. SQLCODE는 "7.3. SQLCA"에서 설명할 SQLCA에 포함되며, WHENEVER 문장에서 검토하는 값도 SQLCODE 값이다.

7.2.4. 사용 예제

다음은 상태 변수를 사용하는 예제 프로그램이다.

```
#include <stdio.h>
#include <sqlca.h>
#include <string.h>
#define USERPASS"tibero/tmax"
int main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userpass[20] = { strlen(USERPASS), USERPASS };
    long SQLCODE = 0;
    VARCHAR ename[24];
    int salary;
    VARCHAR addr[32];
    int empno;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT :userpass;
    printf("Connected.\n");

    printf("Enter the number of employee to show: ");
    scanf("%d", &empno);

    EXEC SQL SELECT ENAME, SALARY, ADDR
        INTO :ename, :salary, :addr
        FROM EMP
        WHERE EMPNO = :empno;

    if (SQLCODE == 0) {                                ... ① ...
        printf("name = %.s, salary = %d, addr = %.s\n",
            ename.len, ename.arr, salary, addr.len, addr.arr);
    }

    if (SQLCODE == 1403) {                                ... ② ...
        printf("no employee found.\n");
    }

    if (SQLCODE < 0) {                                    ... ③ ...
        printf("ERROR: %d\n", sqlca.sqlcode);
    }

    EXEC SQL COMMIT WORK RELEASE;
}
```

상태 변수를 검토하는 코드는 SQL 문장을 실행한 바로 다음에 와야 하며 모든 경우에 대비해야 한다.

위의 예에서는 SQLCODE의 값에 따라 다음과 같이 처리하는 방법이 다르다.

- ① SQLCODE의 값이 0이면 정상적으로 `tbESQL` 문장을 실행한 경우이다.
- ② SQLCODE의 값이 1403이면 결과 로우가 없는 경우이다.
- ③ SQLCODE의 값이 0보다 작으면 에러가 발생한 경우이다.

7.3. SQLCA

SQLCA는 SQL 통신 영역(SQL Communication Area)을 줄여서 부르는 말로, 임의의 SQL 문장이 실행된 결과가 저장되는 구조체 변수이다.

7.3.1. SQLCA 구조체 변수

SQLCA는 헤더 파일 `sqlca.h` 내부에 **sqlca 구조체 변수**로 정의되어 있다. 구조체 변수 `sqlca`에 포함된 데이터는 바로 전에 실행된 SQL 문장의 정보이다. 따라서 다른 SQL 문장을 실행하면 이전에 SQL 문장을 실행한 결과는 더는 `sqlca`에 저장되지 않는다.

이 변수에 포함되는 정보는 다음과 같다.

포함 정보	설명
에러 코드	에러 코드는 앞 절에서 설명한 SQLCODE와 같은 값을 갖는다. 이 값은 WHENEVER 문장에서 참조하는 값이다.
에러 메시지	에러 메시지는 최대 70bytes의 길이를 갖는다.
처리된 로우의 개수 및 파싱 에러의 위치	처리된 로우의 개수는 바로 전에 실행된 SQL 문장에 따라 의미가 달라진다. – SELECT : 질의 결과 반환된 결과 로우의 누적 개수이다. – INSERT : 삽입된 로우의 개수이다. – UPDATE : 갱신된 로우의 개수이다. – DELETE : 삭제된 로우의 개수이다. 파싱 에러의 위치는 SQL 문장을 파싱하는 과정에서 에러가 발생한 경우에 SQL 문장 내의 에러 위치를 가리킨다.
경고 플래그	경고 플래그는 경고 상황이 발생한 경우에 설정된다. 경고 상황으로는 출력 변수에 저장된 컬럼 값이 잘려진 값인 경우, SELECT 또는 FETCH 문장 내의 INTO 절에 포함된 출력 변수의 개수가 결과 로우의 개수에 비하여 적은 경우 등이 있다.

다음은 sqlca 구조체 변수가 정의된 내용이다.

```
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 sqltext[8];
};
```

다음은 sqlca 구조체 변수의 내부에 정의된 각 멤버 변수에 대한 설명이다.

멤버 변수	설명
sqlcaid	SQLCA라는 것을 나타내기 위한 변수이며, 항상 "SQLCA" 문자열을 갖는다.
sqlabc	sqlca 구조체 변수의 Byte 길이가 저장된다.
sqlcode	결과가 숫자로 저장된다. 실행이 성공한 경우 0이며, 이외의 경우 적절한 에러 값 등을 갖는다. 자세한 내용은 "7.2. 상태 변수" 를 참고한다.
sqlerrm	에러 메시지의 Byte 길이가 저장되는 변수이다. 실제 메시지는 sqlerrm.sqlerrmc에 저장되어 있으며, 이 문자열은 마지막 NULL 문자를 포함하지 않으므로 에러 메시지의 Byte 길이를 가지는 sqlerrm.sqlerrml 변수를 반드시 참조해야 한다.
sqlerrp	SQL-99 표준에 있지만, 현재 Tibero에서는 사용하지 않는다.
sqlerrd	sqlerrd[2]와 sqlerrd[4]만 사용된다. sqlerrd[2]는 처리된 로우의 개수, sqlerrd[4]는 파싱 에러의 위치를 나타낸다.
sqlwarn	경고 플래그이다. Byte 하나당 하나의 경고 상황이 발생했음을 나타낸다. 문자 'W'를 저장함으로써 플래그를 설정한다. 다음은 멤버 변수 sqlwarn에 저장되는 각 경고 플래그의 의미이다. – sqlwarn[0] : sqlwarn[1], sqlwarn[2], sqlwarn[3] 중 하나가 설정되면 동시에 이 플래그도 'W'로 설정된다.

멤버 변수	설명
	– sqlwarn[1] : 문자열 출력 변수에 잘린 값이 저장된 경우이다. – sqlwarn[2] : SUM, AVG 등의 집단 함수를 계산할 때 NULL 값이 사용되지 않는 경우이다. – sqlwarn[3] : SELECT 또는 FETCH 문장의 INTO 절에 있는 출력 변수의 개수가 컬럼의 개수와 일치하지 않는 경우이다. – sqlwarn[4] ~ sqlwarn[7] : SQL-99 표준에 있지만, 현재 Tibero에서는 사용하지 않는다.
sqlext	SQL-99 표준에 있지만, 현재 Tibero에서는 사용하지 않는다.

7.3.2. 사용 예제

SQLCA의 변수 sqlca는 전역 변수이며 헤더 파일 sqlca.h 내에 정의되어 있다. 따라서 이 변수를 사용하기 위해서는 다음의 코드를 tbESQL/C 프로그램 처음에 삽입해 주어야 하며, 따로 변수를 선언할 필요는 없다.

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

본 예제는 “7.2.4. 사용 예제”에서 제시한 프로그램을 SQLCA 변수와 출력 배열 변수를 이용하도록 수정한 것이다. 상태 변수 SQLCODE는 구조체 변수 sqlca 내에 포함되어 있으며, SELECT 실행 결과 로우의 개수와 에러 메시지도 sqlca에서 얻을 수 있다.

```
#include <stdio.h>
#include <sqlca.h>
#include <string.h>
#define USERPASS"tibero/tmax"
int main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userpass[20] = { strlen(USERPASS), USERPASS };
    int sqlcode = 0;
    VARCHAR ename[50][24];
    int salary[50];
    VARCHAR addr[50][32];
    int deptno;
    EXEC SQL END DECLARE SECTION;

    int i, count;

    EXEC SQL CONNECT :userpass;
    printf("Connected.\n");
```

```

printf("Enter the number of department to show: ");
scanf("%d", &deptno);

EXEC SQL SELECT ENAME, SALARY, ADDR
      INTO :ename, :salary, :addr
      FROM EMP
      WHERE DEPTNO = :deptno;
sqlcode = sqlca.sqlcode;

if (sqlcode == 0) {
    count = sqlca.sqlerrd[2];
    for (i = 0; i < count; i++) {
        printf("name = %.*s, salary = %d, addr = %.*s\n",
            ename[i].len, ename[i].arr, salary[i], addr[i].len, addr[i].arr);
    }
}

if (sqlcode == +100) {
    printf("no employee found.\n");
}

if (sqlcode < 0) {
    printf("ERROR: %.*s\n",
        sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
}

EXEC SQL COMMIT WORK RELEASE;

return 1;
}

```

7.4. WHENEVER

WHENEVER 문장은 다른 `tbESQL/C` 문장처럼 어떠한 동작을 실행하기 위한 문장이 아니라, 특정 조건이 발생하면 특정 동작을 수행하라는 선언을 위한 문장이다.

7.4.1. 구성요소

WHENEVER 문장은 다음과 같이 조건 부분과 동작 부분으로 구성된다.

```
EXEC SQL WHENEVER {조건} {동작};
```

- {조건}

WHENEVER 문장의 조건 부분에 대한 설명은 다음과 같다.

항목	설명
SQLERROR	SQLCODE가 음수인 경우에 해당된다.
SQLWARNING	SQLCODE가 0보다 큰 경우 또는 SQLCA 내의 경고 플래그가 설정된 경우이다. SQLCODE가 NOT FOUND 코드 값(+100)에 해당되는 경우는 제외된다. 경고 플래그가 설정된 경우는 sqlca.sqlwarn[0]에 'W' 값이 저장된 경우이다.
NOT FOUND	SQLCODE가 NOT FOUND 코드 값인 경우에 해당된다. 질의의 결과 로우가 없거나 커서를 이용해 더 이상 액세스할 결과 로우가 없거나 INSERT, UPDATE, DELETE 문장에 의해 처리된 로우가 없을 때 NOT FOUND 경고가 발생한다.

- {동작}

WHENEVER 문장의 동작 부분에 대한 설명은 다음과 같다.

항목	설명
CONTINUE	프로그램의 다음 문장부터 계속하여 진행한다. tbESQL/C 프로그램에서 에러 또는 경고가 발생했을 때 디폴트로 CONTINUE를 실행한다. 이는 에러 또는 경고에 대한 처리가 없는 것과 마찬가지다.
DO	특정 함수를 호출한다. 호출된 함수는 대개 에러 처리 함수이며, 함수가 실행되고 나면 에러 또는 경고가 발생한 다음 문장부터 실행이 계속된다. 만약 치명적인 에러가 발생했다면 함수 내에서 트랜잭션을 롤백시키고 프로그램을 종료할 수도 있다.
DO BREAK	루프 내에서 에러 또는 경고가 발생했을 때 BREAK를 실행한다. 만약 루프 내에서 사용되지 않았다면 컴파일 에러를 발생시킨다.
DO CONTINUE	루프 내에서 에러 또는 경고가 발생했을 때 CONTINUE를 실행한다. 만약 루프 내에서 사용되지 않았다면 컴파일 에러를 발생시킨다.
GOTO	GOTO를 실행하여 프로그램 내의 특정 위치로 분기한다. 분기할 위치는 레이블로 정의한다.
STOP	프로그램을 종료하며 현재 트랜잭션을 롤백한다. 에러 메시지의 반환 없이 바로 종료된다.

다음은 WHENEVER 문장을 사용하는 예이다.

```
EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

위의 예에서 NOT FOUND가 조건, DO BREAK가 동작에 해당된다.

7.4.2. 사용 예제

WHENEVER 문장이 선언되면 그 효과는 바로 다음 문장에서부터 시작된다. **tbESQL/C** 프로그램은 **SQL** 문장을 실행할 때마다 **WHENEVER** 문장에 포함된 조건이 발생하였는지 자동적으로 검토한다. 만약 조건이 발생하면 선언된 동작을 수행한다.

WHENEVER 문장의 효과는 같은 조건에 대한 다음 **WHENEVER** 문장이 나타날 때까지 이어진다. **WHENEVER** 문장이 나타나더라도 선언된 조건이 다르면 이전의 **WHENEVER** 문장의 효과는 지속된다. 즉, **WHENEVER** 문장의 효과는 블록 구조와는 상관이 없다.

WHENEVER 문장을 사용할 때는 다음과 같은 사항에 유의해야 한다.

- **NOT FOUND**가 **SELECT**, **FETCH** 문장에 의한 것인지 **INSERT**, **UPDATE**, **DELETE** 문장에 의한 것인지 구분해야 한다.

INSERT, **UPDATE**, **DELETE** 문장은 **SELECT** 또는 **FETCH** 문장과 마찬가지로 처리된 로우가 하나도 없으면 **NOT FOUND**를 발생한다. 이때 프로그램 내에서는 **NOT FOUND**가 **SELECT**, **FETCH** 문장에 의한 것인지 **INSERT**, **UPDATE**, **DELETE** 문장에 의한 것인지 구분해 주어야 한다.

다음은 **INSERT** 문장을 잘못 사용하여 의도하던 바와 다르게 실행될 수 있는 경우의 예이다.

[예 7.1] 잘못된 INSERT 문장

```
EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor INTO ...
    ...
    EXEC SQL INSERT INTO ...
}
```

위의 코드를 작성한 의도는 모든 결과 로우를 액세스하여 더 이상의 결과 로우가 없을 때 **FETCH** 문장에서 **NOT FOUND**를 발생하면 루프를 빠져 나가는 것이다. 하지만 **INSERT** 문장에서 삽입되는 로우가 없다면 마찬가지로 **NOT FOUND**를 발생하고 루프를 빠져나가게 된다. 따라서 모든 결과 로우를 액세스하지 못하게 된다.

다음은 [\[예 7.1\]](#)에서 발생한 문제를 해결한 예이다.

```
long SQLCODE = 0;
...
while (1)
{
    EXEC SQL FETCH cursor INTO ...
    if (SQLCODE == +100) break;
    ...
    EXEC SQL INSERT INTO ...
}
```

```

    if (SQLCODE == +100) ...
}

```

위의 예에서는 WHENEVER 문장을 사용하지 않고 직접 SQLCODE를 검토하는 코드를 삽입하였다.

- DO 또는 GOTO 동작을 선언했을 때 잘못하면 무한 루프에 빠질 수 있다.

WHENEVER 문장에서 DO 또는 GOTO 동작을 선언했을 때 에러 또는 경고를 처리하기 위한 루틴에서 또다시 에러 또는 경고가 발생하면 무한 루프에 빠질 수 있다.

다음은 GOTO 동작을 선언한 경우에 무한 루프가 발생할 수 있는 예이다.

[예 7.2] GOTO가 선언된 경우의 무한 루프

```

EXEC SQL WHENEVER SQLERROR GOTO error_handle;
EXEC SQL SELECT ...
...
error_handle:
EXEC SQL DELETE ...

```

위의 예에서 SELECT 문장을 실행하는 중에 에러가 발생하면 error_handle 레이블로 분기하고, DELETE 문장을 실행한다. 이때 DELETE 문장에서 에러가 발생한다면 DELETE 문장은 무한 반복 상태가 된다.

다음은 [예 7.2]의 무한 루프 문제를 해결한 예이다.

```

EXEC SQL WHENEVER SQLERROR GOTO error_handle;
EXEC SQL SELECT ...
...
error_handle:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL DELETE ...

```

위의 예에서는 에러 처리 부분의 맨 앞에 WHENEVER 문장을 첨가하여 에러가 발생하더라도 계속 실행하도록 하여 무한 루프가 발생할 가능성을 없앴다.

다음과 같이 코드를 작성해도 [예 7.2]에서 발생하는 문제를 해결할 수 있다.

```

/* 메인 루틴 */
EXEC SQL WHENEVER SQLERROR DO error_handle(SQLCODE);
...
/* 에러 처리 함수 */
int error_handle(long SQLCODE)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    ...
    EXEC SQL WHENEVER SQLERROR DO error_handle(SQLCODE);
}

```

위의 예에서는 WHENEVER 문장에 DO 동작을 선언해 함수를 호출하였다. 함수의 맨 처음에 WHENEVER 문장을 삽입하고 맨 마지막에 에러 처리 함수를 호출하는 WHENEVER 문장을 다시 삽입하였다.

- **GOTO** 동작을 선언하는 경우 분기할 레이블의 위치를 SQL 문장에서 액세스할 수 있는 범위에 두어야 한다.

즉, SQL 문장에서 참조할 수 있는 레이블로 선언해야 한다. **WHENEVER** 문장이 선언된 **tbESQL/C** 프로그램을 프리컴파일하면 모든 SQL 문장 뒤에 **SQLCODE**를 검토하고 선언된 동작을 수행하는 코드를 삽입한다. 만약 **GOTO** 동작에서 분기할 레이블이 현재 삽입된 코드에서 참조할 수 없는 위치에 있다면 컴파일 에러가 발생한다.

다음은 **GOTO** 레이블에 의하여 컴파일 에러가 발생하는 예이다.

```
/* 메인 루틴 */
EXEC SQL WHENEVER SQLERROR GOTO error_handle:
...
error_handle:
...
/* 임의의 함수 */
int f()
{
    EXEC SQL SELECT ...
    ...
}
```

위의 예에서 메인 루틴 내에서는 **error_handle** 레이블을 참조할 수 있지만, 함수 **f**에서는 참조할 수 없다. 이러한 문제를 해결하기 위하여 함수 **f** 내에 새로운 **WHENEVER** 문장을 삽입하는 방법도 있다.

제8장 tbESQL/C 문장

본 장에서는 tbESQL/C 프로그램에서 데이터베이스 처리를 위해 사용하는 tbESQL/C 문장을 설명한다.

8.1. 개요

tbESQL/C 프로그램은 C 프로그래밍 언어의 소스 코드와 tbESQL/C 문장이 혼합되어 있다. **tbESQL/C 문장**(tbESQL/C Statement)은 tbESQL/C 프로그램 내에서 SQL 질의, 로우의 삽입과 갱신, 제거 등과 같은 데이터베이스와 관련된 처리를 하는 문장을 말한다.

tbESQL/C 문장은 크게 다음과 같이 두 가지로 구성된다.

- **실행 문장**(Executable Statements)

실행 문장은 말 그대로 데이터베이스에 어떠한 동작을 실행하기 위한 문장이다.

- **지시어**(Directive)

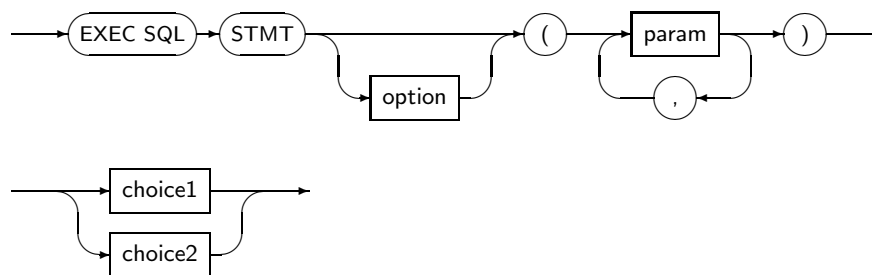
지시어는 tbESQL/C 프로그램의 실행을 설정하기 위한 것으로 데이터베이스에 대한 실행은 이루어지지 않는다.

8.2. tbESQL/C 문장 문법

다음은 tbESQL/C 문장의 문법을 나타내는 그림이다.

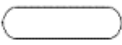
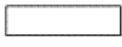
[그림 8.1] tbESQL/C 문장의 문법

esql_stmt



[그림 8.1]을 기준으로 tbESQL/C 문장의 문법을 해석하는 방법은 다음과 같다.

항목	설명
<i>esql_stmt</i>	tbESQL/C 문장의 문법을 대표하는 이름은 왼쪽 위에 나타낸다. (예: <i>esql_stmt</i>)

항목	설명
	타원 안의 포함되는 문자는 키워드(Keyword)이며, 반드시 tbESQL/C 문장 내에 포함되어 있어야 한다. (예: EXEC SQL, STMT)
	사각형 안의 문자는 tbESQL/C 문장에 포함된 문법의 구성요소이며, 구성요소에 맞는 적절한 문자열로 바꾸어야 한다. (예: option, param, choice1, choice2)
	원 안의 문자는 tbESQL/C 문장의 기호이며, 반드시 tbESQL/C 문장 내에 포함되어 있어야 한다. (예: 소괄호(()), 콤마(,))
	tbESQL/C 문장을 완성하는 순서는 화살표를 따라가면 된다. 여러 갈래로 갈라지면서 순방향으로 이동하는 화살표는 두 가지 형태로 된다. 이는 [그림 8.1] 에서 option은 포함되거나 포함되지 않은 경우이며, choice1, choice2는 여러 가지 중에 하나만이 tbESQL/C 문장에 포함되어야 하는 경우이다. 또는 역방향으로 이동하는 화살표는 포함되지 않거나 한번 이상 포함되는 경우이다. [그림 8.1] 에서 콤마(,)에 해당되며, param은 반드시 한번 이상 포함되어야 한다.

다음의 tbESQL/C 문장은 [\[그림 8.1\]](#)에 따라 모두 유효한 예이다.

```
EXEC SQL STMT (param) choice1;

EXEC SQL STMT option (param, param) choice1;

EXEC SQL STMT (param, param) choice2;
```

8.3. tbESQL/C 문장 공통 문법

본 절에서는 tbESQL/C 문장에서 공통적으로 자주 사용되는 부분을 설명한다.

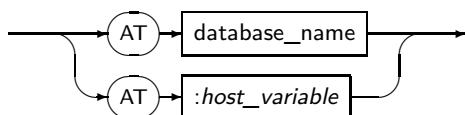
8.3.1. AT 절

AT 절은 이름이 있는 데이터베이스 연결을 사용해 tbESQL/C 문장을 수행할 때 사용한다.

AT의 세부 내용은 다음과 같다.

- 문법

at_clause



- 구성요소

구성요소	설명
database_name	사용할 데이터베이스의 이름을 명시한다. 여러 개의 데이터베이스 접속을 구분하여 관리하고 싶을 때 database_name을 사용한다. 데이터베이스의 이름은 DECLARE DATABASE를 사용해 미리 선언되어 있어야 한다. 만약 선언되어 있지 않은 이름을 사용할 경우 에러가 발생한다.
:host_variable	사용할 데이터베이스의 이름이 저장된 호스트 변수를 명시한다. 데이터베이스의 이름은 미리 선언되어 있어야 한다.

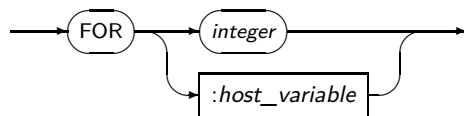
8.3.2. FOR 절

FOR 절은 `tbESQL/C` 문장을 반복해서 수행할 필요가 있을 때 사용한다.

FOR 절의 세부 내용은 다음과 같다.

- 문법

for_clause



- 구성요소

구성요소	설명
integer	반복 횟수를 명시한다.
:host_variable	반복 횟수가 저장된 호스트 변수를 명시한다. 호스트 변수는 int 등의 숫자를 저장할 수 있는 타입이면 된다.

8.3.3. DESCRIPTOR 이름

DESCRIPTOR 이름은 Dynamic SQL을 사용할 때 필요한 DESCRIPTOR를 가리키는 이름이다.

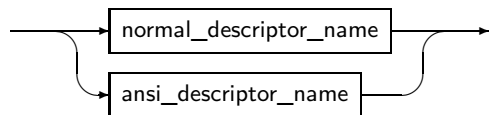
DESCRIPTOR 이름은 프리컴파일러 옵션 즉 `MODE`, `DYNAMIC`의 값에 따라 다음과 같이 사용될 수 있다.

옵션 값	설명
ANSI	ansi_descriptor_name이 사용된다.
ANSI 외	normal_descriptor_name이 사용된다.

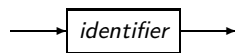
DESCRIPTOR 이름의 세부 내용은 다음과 같다.

- 문법

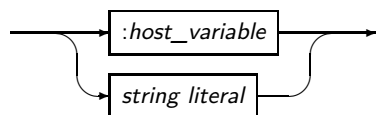
descriptor_name



normal_descriptor_name



ansi_descriptor_name



- 구성요소

- descriptor_name

구성요소	설명
normal_descriptor_name	MODE가 TIBERO(ORACLE)일 때 사용한다.
ansi_descriptor_name	MODE가 ANSI(ISO)일 때 사용한다.

- normal_descriptor_name

구성요소	설명
identifier	서술자의 이름을 정의한 식별자이다.

- ansi_descriptor_name

구성요소	설명
:host_variable	서술자의 이름이 저장된 호스트 변수를 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등의 문자열을 저장할 수 있는 타입이다.
string literal	서술자의 이름을 작은따옴표(' ')로 감싸서 사용한다.

8.4. tbESQL/C 문장 목록

본 절에서는 Tibero에서 제공하는 tbESQL/C 문장을 알파벳 순으로 설명한다. 단, 각 tbESQL/C 문장의 구성요소를 설명할 때 키워드는 특별한 경우가 아니면 설명하지 않는다.

다음은 tbESQL/C 문장을 요약한 목록이다.

- 실행문장

tbESQL/C 문장	설명
ALLOCATE	PL/SQL 블록에서 커서변수를 사용하기 위해 커서를 할당한다.
ALLOCATE DESCRIPTOR	서술자에 메모리를 할당한다.
CALL	저장된 프러시저를 호출하여 해당 프러시저를 수행한다.
CLOSE	커서를 닫고 더 이상 사용하지 않는다.
COMMIT	트랜잭션을 커밋한다.
CONNECT	Tibero의 데이터베이스에 접속한다.
CONTEXT ALLOCATE	컨텍스트에 메모리를 할당한다.
CONTEXT FREE	컨텍스트에 할당된 메모리를 해제한다.
CONTEXT USE	지정된 컨텍스트를 사용한다.
DEALLOCATE DESCRIPTOR	서술자에 할당된 메모리를 해제한다.
DELETE	로우를 삭제한다.
EXECUTE	준비된 Dynamic SQL 문장을 실행한다.
EXECUTE DESCRIPTOR	준비된 Dynamic SQL 문장을 실행한다. (ANSI)
EXECUTE ... END-EXEC	PL/SQL 블록을 실행한다.
EXECUTE IMMEDIATE	Dynamic SQL 문장을 바로 실행한다.
FETCH	커서를 이용하여 다음 로우를 읽는다.
FETCH DESCRIPTOR	커서를 이용하여 다음 로우를 읽는다. (ANSI)
GET DESCRIPTOR	지정한 서술자에서 원하는 정보를 가져온다.
INSERT	로우를 삽입한다.
LOB CLOSE	오픈된 LOB 대용량 객체를 닫는다.
LOB CREATE TEMPORARY	TEMPORARY LOB을 생성한다.
LOB DESCRIBE	대용량 객체형의 데이터 타입의 속성을 얻어 온다.
LOB FREE TEMPORARY	생성된 TEMPORARY LOB의 메모리를 해제한다.
LOB OPEN	LOB 대용량 객체를 오픈한다.
LOB READ	LOB 지시자(LOB locator)를 사용하여 내용을 읽는다.
LOB WRITE	LOB 지시자가 가리키는 위치에 내용을 기록한다.

tbESQL/C 문장	설명
OPEN	커서를 열고 연관된 SQL 문장을 실행한다.
PREPARE	Dynamic SQL 문장을 준비한다.
ROLLBACK	트랜잭션에 롤백을 수행한다.
SAVEPOINT	저장점(Savepoint)을 설정한다.
SELECT	SQL 질의를 수행한다.
UPDATE	로우를 갱신한다.

- 지시어

tbESQL/C 문장	설명
DECLARE CURSOR	SQL 문장과 연관된 커서를 선언한다.
DECLARE DATABASE	새로운 데이터베이스 접속을 선언한다.
DECLARE STATEMENT	SQL 문장이나 PL/SQL 문장을 identifier를 통해 선언한다.
DESCRIBE	서술자를 초기화한다.
DESCRIBE DESCRIPTOR	서술자에 호스트 변수의 정보를 저장한다.
ENABLE THREADS	멀티 스레드를 사용한다.
SET DESCRIPTOR	지정한 서술자에 사용자가 입력해야 할 정보를 쓴다.
TYPE	사용자가 지정한 데이터 타입을 외부 데이터 타입으로 동격화한다.
VAR	특정 호스트 변수의 데이터 타입을 외부 데이터 타입으로 동격화한다.
WHENEVER	에러가 발생했을 때 해당 에러에 대한 처리 방법을 지정한다.

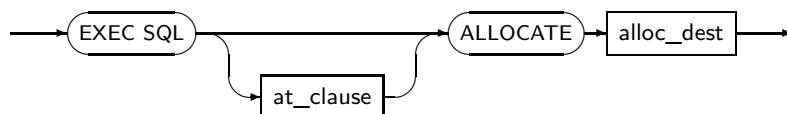
8.4.1. ALLOCATE

ALLOCATE는 PL/SQL 블록에서 cursor variable을 사용하기 위한 문장이다. cursor variable은 ALLOCATE 문장을 사용하기 전에 SQL_CURSOR를 통해 선언되어야 한다.

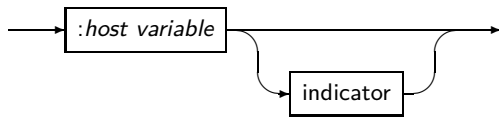
ALLOCATE의 세부 내용은 다음과 같다.

- 문법

allocate_statement



alloc_dest



- 구성요소

- allocate_statement

구성요소	설명
at_clause	커밋을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
alloc_dest	커서를 할당할 변수를 명시한다.

- alloc_dest

구성요소	설명
:host_variable	커서 변수를 저장하고 있는 호스트 변수이다.
indicator	지시자 변수를 명시할 때 사용한다.

- 예제

다음은 ALLOCATE DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;  
    SQL_CURSOR csr_var;  
EXEC SQL END DECLARE SECTION;  
  
EXEC SQL ALLOCATE :csr_var;
```

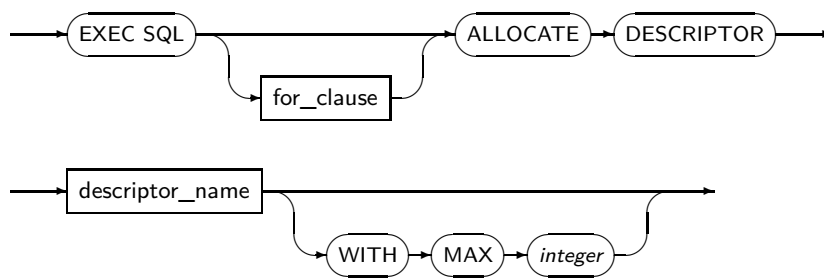
8.4.2. ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR는 서술자에 메모리를 할당할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

ALLOCATE DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

allocate_descriptor_statement



- 구성요소

구성요소	설명
for_clause	반복 횟수를 지정한다. 자세한 내용은 “8.3.2. FOR 절” 을 참고한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름” 을 참고한다.
WITH MAX integer	사용할 호스트 변수의 최대 개수를 지정한다. (기본값: 100)

- 예제

다음은 ALLOCATE DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL ALLOCATE DESCRIPTOR 'input_descriptor';

EXEC SQL ALLOCATE DESCRIPTOR 'output_descriptor';
```

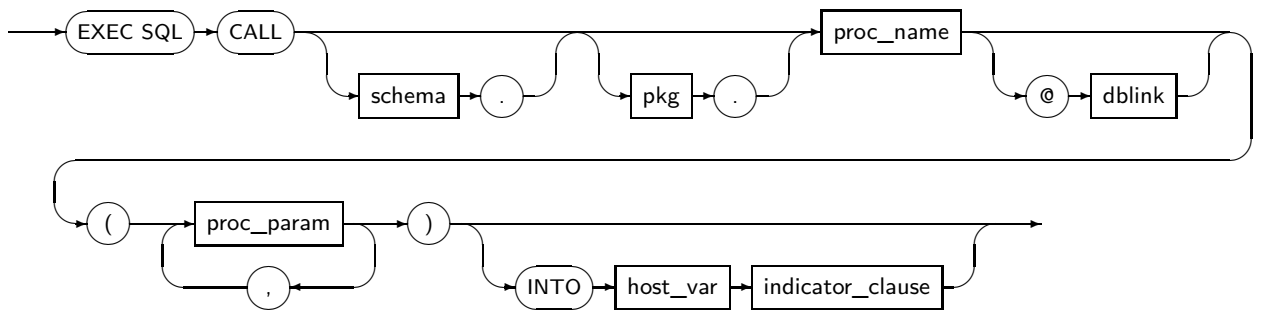
8.4.3. CALL

CALL은 저장된 프러시저를 호출하여 해당 프러시저를 수행하는 문장이다. 이 문장을 사용하기 위해선 먼저 프러시저가 저장된 데이터베이스에 접속한 상태여야 한다.

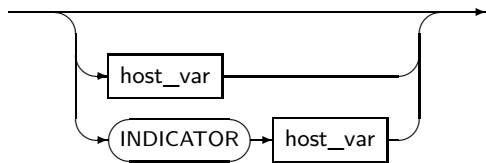
CALL의 세부 내용은 다음과 같다.

- 문법

call_statement



indicator_clause



- 구성요소

– call_statement

구성요소	설명
schema	프러시저를 가지고 있는 스키마를 명시한다.
pkg	프러시저가 저장된 패키지를 명시한다.
dblink	프러시저가 위치한 database link 이름을 명시한다.
proc_param	프러시저의 input으로 쓰이는 파라미터를 명시한다.
host_var	프러시저의 return value를 저장할 host variable을 명시한다.
indicator_clause	indicator를 명시한다.

– indicator_clause

구성요소	설명
host_var	indicator 정보를 저장할 host variable을 명시한다.

- 예제

다음은 CALL를 사용하는 예이다.

```
int hv_a = 1;
int hv_b = 2;
int hv_result;

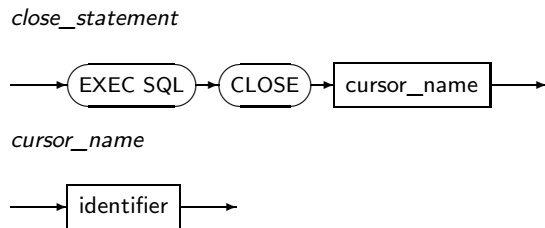
EXEC SQL CALL get_result_sum( :hv_a, :hv_b ) INTO :hv_result;
```

8.4.4. CLOSE

CLOSE는 커서를 닫을 때 사용하는 문장이다. 이 문장은 현재 열려 있는 커서에만 사용할 수 있다. 커서를 닫으면 커서를 생성할 때 할당되었던 모든 시스템 리소스가 반환된다.

CLOSE의 세부 내용은 다음과 같다.

- 문법



- 구성요소

– close_statement

구성요소	설명
cursor_name	닫으려는 커서의 이름을 명시한다.

– cursor_name

구성요소	설명
identifier	커서의 이름을 정의한 식별자이다.

- 예제

다음은 CLOSE를 사용하는 예이다.

```
EXEC CLOSE emp_cursor;
```

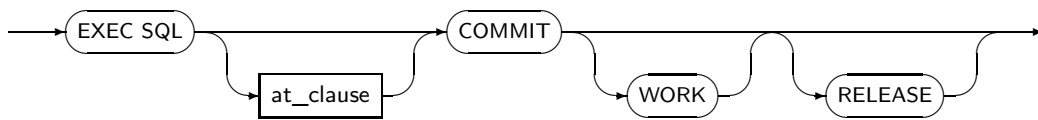
8.4.5. COMMIT

COMMIT은 현재 진행 중인 트랜잭션을 커밋하고 트랜잭션에 의해 갱신된 모든 내용을 데이터베이스에 반영할 때 사용하는 문장이다.

COMMIT의 세부 내용은 다음과 같다.

- 문법

commit_statement



- 구성요소

구성요소	설명
at_clause	커밋을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
WORK	기존 ESQL 프로그램과의 호환성을 위한 키워드로 특별한 의미는 없다.
RELEASE	모든 리소스를 반환하고 데이터베이스 접속을 종료한다.

- 예제

다음은 COMMIT을 사용하는 예이다.

```
EXEC SQL COMMIT;

EXEC SQL COMMIT WORK RELEASE;
```

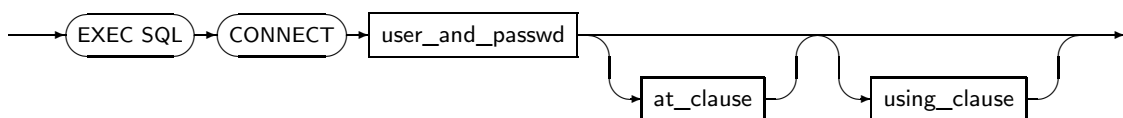
8.4.6. CONNECT

CONNECT는 데이터베이스에 접속할 때 사용하는 문장이다. 이 문장을 사용할 때는 사용자 이름과 패스워드를 반드시 명시해야 한다. 데이터베이스 관리자(DBA: Database Administrator, 이하 DBA)로 접속할 수도 있다.

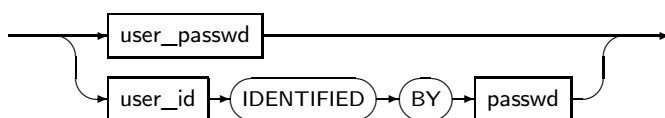
CONNECT의 세부 내용은 다음과 같다.

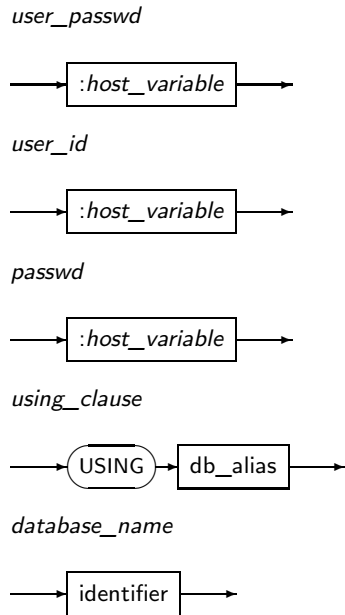
- 문법

connect_statement



user_and_passwd





- 구성요소

- connect_statement

구성요소	설명
user_and_passwd	사용자 이름과 패스워드를 명시한다.
at_clause	접속할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
using_clause	데이터베이스의 별칭을 명시한다.

- user_and_passwd

구성요소	설명
user_password	사용자 이름과 패스워드를 명시한다. 사용자 이름과 패스워드의 중간에 슬래시(/)를 포함해 하나의 문자열로 표현한다. (예: 'tibero/tibero')
user_id	사용자 이름을 명시한다. 호스트 변수이며, 문자열이 직접 올 수 없다.
passwd	패스워드를 명시한다. 호스트 변수이며, 문자열이 직접 올 수 없다.

- user_passwd

구성요소	설명
:host_variable	사용자 이름과 패스워드를 저장하고 있는 호스트 변수이다.

– user_id

구성요소	설명
:host_variable	사용자의 계정을 저장하고 있는 호스트 변수이다.

– passwd

구성요소	설명
:host_variable	패스워드를 저장하고 있는 호스트 변수이다.

– using_clause

구성요소	설명
db_alias	데이터베이스의 별칭을 명시한다.

– database_name

구성요소	설명
identifier	데이터베이스의 이름을 정의한 식별자이다.

- 예제

다음은 CONNECT를 사용하는 예이다.

```
EXEC SQL CONNECT :user IDENTIFIED BY :password;
```

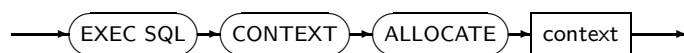
8.4.7. CONTEXT ALLOCATE

CONTEXT ALLOCATE는 tbESQL/C 문장의 실행 정보를 담고 있는 컨텍스트를 위한 메모리를 할당할 때 사용하는 문장이다.

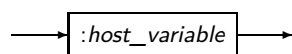
CONTEXT ALLOCATE의 세부 내용은 다음과 같다.

- 문법

context_allocate_statement



context



- 구성요소

– context_allocate_statement

구성요소	설명
context	할당할 컨텍스트 변수이다. sql_context 타입의 포인터로 미리 선언해둔 호스트 변수여야 한다.

– context

구성요소	설명
:host_variable	sql_context 타입의 포인터로 선언한 호스트 변수이다.

- 예제

다음은 CONTEXT ALLOCATE를 사용하는 예이다.

```
EXEC SQL CONTEXT ALLOCATE :ctx1;
```

8.4.8. CONTEXT FREE

CONTEXT FREE는 컨텍스트에 할당된 메모리를 해제할 때 사용하는 문장이다.

CONTEXT FREE의 세부 내용은 다음과 같다.

- 문법

context_free_statement



- 구성요소

구성요소	설명
context	해제할 컨텍스트 변수이다. sql_context 타입의 포인터로 미리 선언되어 있어야 하며, CONTEXT ALLOCATE를 통해 이미 할당되어 있어야 한다.

- 예제

다음은 CONTEXT FREE를 사용하는 예이다.

```
EXEC SQL CONTEXT FREE :ctx1;
```

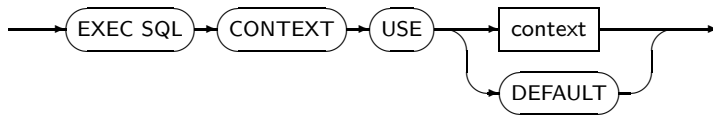
8.4.9. CONTEXT USE

CONTEXT USE는 여러 개의 컨텍스트 중에서 사용하려는 컨텍스트를 지정할 때 사용하는 문장이다.

CONTEXT USE의 세부 내용은 다음과 같다.

- 문법

context_use_statement



- 구성요소

구성요소	설명
context	사용할 컨텍스트 변수를 명시한다. sql_context 타입의 포인터로 미리 선언되어 있어야 하며, CONTEXT ALLOCATE를 통해 이미 할당되어 있어야 한다.
DEFAULT	DEFAULT로 선언할 경우 디폴트 컨텍스트를 사용하게 된다. 디폴트 컨텍스트는 할당과 해제를 할 필요 없이 그대로 사용할 수 있다.

- 예제

다음은 CONTEXT USE를 사용하는 예이다.

```
EXEC SQL CONTEXT USE :ctx1;

EXEC SQL CONTEXT USE DEFAULT;
```

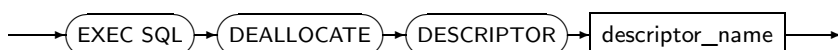
8.4.10. DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR는 ALLOCATE DESCRIPTOR를 사용해 할당된 서술자의 메모리를 해제할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

DEALLOCATE DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

deallocate_descriptor_statement



- 구성요소

구성요소	설명
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

다음은 DEALLOCATE DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL DEALLOCATE DESSCRIPTOR 'input_descriptor';
```

8.4.11. DECLARE CURSOR

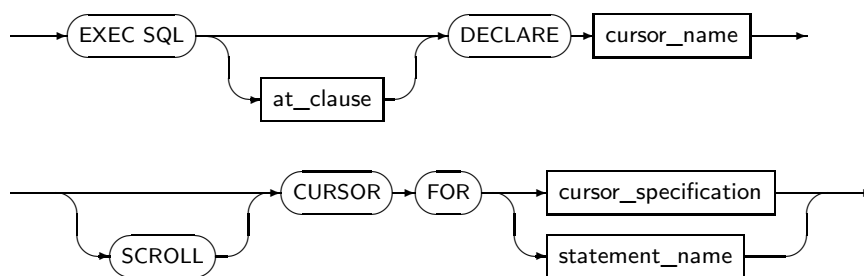
DECLARE CURSOR는 커서를 선언할 때 사용하는 문장이다. 이 문장을 사용해 스크롤 가능 커서(Scrollable Cursor)를 선언할 수도 있다. 커서를 선언할 때는 커서의 이름을 반드시 명시하여야 하며, **SELECT** 문장과 연관시켜야 한다.

Dynamic SQL 문장에 대한 커서인 경우에는 문장 이름(Statement Name)과 연관시킨다. 이때 문장 이름은 **PREPARE**를 통해 준비되어 있어야 한다.

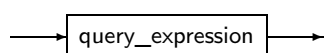
DECLARE CURSOR의 세부 내용은 다음과 같다.

- 문법

declare_cursor_statement



cursor_specification



- 구성요소

– declare_cursor_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
cursor_name	커서의 이름을 명시한다.
SCROLL	스크롤 가능 커서로 선언한다.
cursor_specification	커서로 수행할 문장을 명시한다.
statement_name	준비된 문장의 이름을 명시한다. 사용될 문장은 PREPARE를 통해 준비되어 있어야 한다.

– cursor_specification

구성요소	설명
query_expression	SELECT 문장을 명시한다. INTO 절을 포함할 수 없다.

- 예제

다음은 DECLARE CURSOR를 사용한 예이다.

```
EXEC SQL DECLARE emp_cursor1 CURSOR FOR
    SELECT EMPNO, ENAME, SALARY FROM EMP;

EXEC SQL DECLARE emp_cursor2 SCROLL CURSOR FOR
    SELECT EMPNO, ENAME, SALARY FROM EMP;

EXEC SQL DECLARE dyn_cursor CURSOR FOR dyn_stmt;
```

8.4.12. DECLARE DATABASE

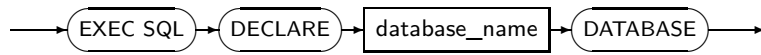
DECLARE DATABASE는 데이터베이스를 선언할 때 사용하는 문장이다.

여러 개의 데이터베이스 접속을 사용할 때 각각의 접속은 데이터베이스의 이름으로 관리된다. DECLARE DATABASE로 선언한 데이터베이스의 이름을 CONNECT 문장이나 COMMIT 문장의 RELEASE를 통해 각각을 구분하여 관리할 수 있다.

DECLARE DATABASE의 세부 내용은 다음과 같다.

- 문법

declare_database_statement



- 구성요소

구성요소	설명
database_name	선언할 데이터베이스 이름을 명시한다.

- 예제

다음은 DECLARE DATABASE를 사용한 예이다.

```
EXEC SQL DECLARE D1 DATABASE;

EXEC SQL CONNECT :user_pass AT D1;
```

8.4.13. DECLARE STATEMENT

DECLARE STATEMENT는 SQL 문장이나 PL/SQL 문장을 identifier로 지정하기 위해 사용하는 문장이다.

DECLARE STATEMENT의 세부 내용은 다음과 같다.

- 문법

declare_statement_statement



- 구성요소

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절” 을 참고한다.
statement_name	준비된 문장의 이름을 명시한다. 이 때의 이름은 identifier로 사용할 수 있다.

- 예제

다음은 DECLARE STATEMENT를 사용한 예이다.

```
EXEC SQL DECLARE exp_stmt STATEMENT;
EXEC SQL PREPARE exp_stmt FROM :query_stmt;
```

8.4.14. DELETE

DELETE는 테이블 또는 뷰에서 로우를 삭제할 때 사용하는 문장이다.

DELETE의 세부 내용은 다음과 같다.

- 문법

DELETE의 문법에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

- 특권

DELETE를 사용하려면, 대상이 되는 테이블 또는 뷰에 대한 DELETE 객체 특권을 갖고 있거나 DELETE ANY TABLE 시스템 특권을 갖고 있어야 한다.

- 구성요소

구성요소	설명
FOR	입력 배열 변수와 함께 사용될 경우 FOR 절을 이용하여 DELETE를 실행할 입력 배열 변수의 크기를 정할 수 있다. FOR 절이 포함되어 있지 않거나 실행할 배열의 크기가 입력 배열 변수의 크기보다 크면, 배열 전체에 대하여 DELETE를 실행한다.
CURRENT OF	DELETE를 커서와 함께 사용할 수도 있다. 현재 커서가 가리키는 로우를 삭제하려면 WHERE 절에 CURRENT OF와 커서 이름을 포함시킨다. 이때 커서는 닫혀 있지 않아야 한다.

- 예제

다음은 DELETE를 사용하는 예이다.

```
EXEC SQL DELETE FROM EMP;

EXEC SQL DELETE FROM EMP WHERE EMPNO = :empno;

EXEC SQL FOR :count DELETE FROM EMP
      WHERE EMPNO = :empno_arr;

EXEC SQL DELETE FROM EMP
      WHERE CURRENT OF emp_cursor;
```

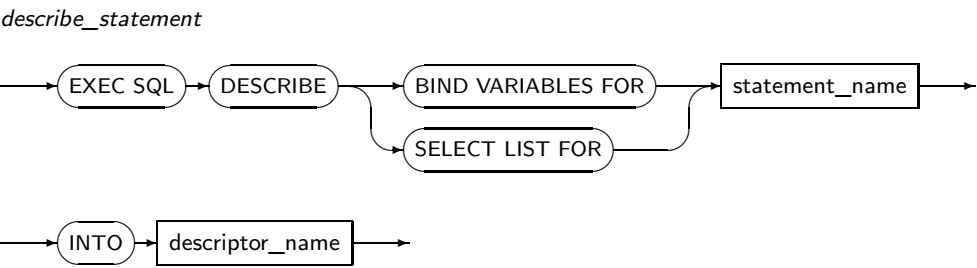
위의 예에서 알 수 있듯이 DELETE 문장은 EXEC SQL로 시작한다는 것을 제외하면, 일반적인 SQL 문장의 문법과 크게 다르지 않다.

8.4.15. DESCRIBE

DESCRIBE는 호스트 변수의 서술자를 초기화할 때 사용하는 문장이다. 이 문장에는 Dynamic SQL 문장이 사용되는데, 사용될 문장은 미리 **PREPARE**를 통해 준비해야 한다.

DESCRIBE의 세부 내용은 다음과 같다.

- 문법



- 구성요소

구성요소	설명
BIND VARIABLES FOR	입력으로 사용되는 호스트 변수의 서술자를 초기화하고, 바인드 변수를 바인딩 하기 위해 사용한다. 이 부분을 명시하지 않으면 SELECT LIST FOR 가 디폴트이다.
SELECT LIST FOR	SELECT 문장의 SELECT 리스트의 정보를 위한 서술자를 초기화한다. 명시하지 않을 경우 기본값이다.
statement_name	사용될 문장의 이름을 명시한다. 사용될 문장의 이름은 이미 PREPARE 를 통해 준비되어 있어야 한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR 를 통해 이미 할당되어 있어야 한다. 자세한 내용은 "8.3.3. DESCRIPTOR 이름" 을 참고한다.

- 예제

다음은 **DESCRIBE**를 사용하는 예이다.

```
EXEC SQL PREPARE pstmt FROM :query_string;

EXEC SQL DECLARE emp_csr
  FOR SELECT empno, ename, sal, comm FROM emp
  WHERE deptno = :dept_no;
```

```
EXEC SQL DESCRIBE BIND VARIABLES FOR pstmt
      INTO bind_descdescriptor;

EXEC SQL OPEN emp_csr
      USING bind_descdescriptor;

EXEC SQL DESCRIBE SELECT LIST FOR pstmt
      INTO select_descdescriptor;

EXEC SQL FETCH emp_csr
      INTO select_descdescriptor;
```

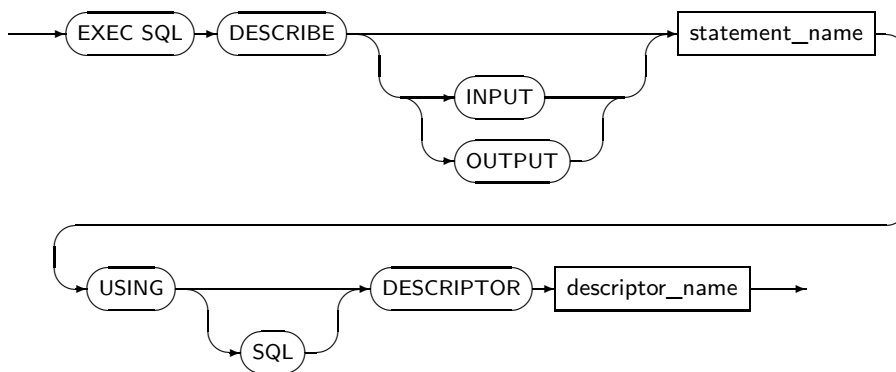
8.4.16. DESCRIBE DESCRIPTOR

DESCRIBE DESCRIPTOR는 서술자에 호스트 변수의 정보를 저장할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

DESCRIBE DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

ansi_describe_statement



- 구성요소

구성요소	설명
INPUT	서술자가 입력과 출력 중에 어느 곳에 사용될지 지정한다. 생략이 가능하며 기본값은 INPUT이다. 사용자가 바인드 변수에 값을 직접 입력하여, tbESQL/C 라이브러리에서 사용될 수 있도록 한다.

구성요소	설명
OUTPUT	서술자가 입력과 출력 중에 어느 곳에 사용될지 지정한다. 생략이 가능하며 기본값은 INPUT이다. 사용자가 바인드 변수의 멤버를 통해 tbESQL/C 라이브러리가 동작하면서 저장된 결과 값의 메타데이터(metadata)를 확인하고, 적합한 대응을 할 수 있도록 해준다.
statement_name	사용될 문장의 이름을 명시한다. 사용될 문장의 이름은 이미 PREPARE 를 통해 준비되어 있어야 한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드로 특별한 의미는 없다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

다음은 **DESCRIBE DESCRIPTOR**를 사용하는 예이다.

```
EXEC SQL DESCRIBE INPUT S USING DESCRIPTOR 'input_descriptor';

EXEC SQL DESCRIBE OUTPUT S USING DESCRIPTOR 'output_descriptor';
```

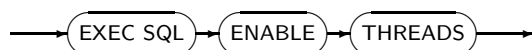
8.4.17. ENABLE THREADS

ENABLE THREADS는 멀티 스레드를 사용하기 전에 반드시 호출되어야 하는 문장이다. 이 문장을 사용하지 않고 컨텍스트 관련 문장을 사용할 경우 예러가 발생한다.

ENABLE THREADS의 세부 내용은 다음과 같다.

- 문법

enable_threads_statement



- 예제

다음은 **ENABLE THREADS**를 사용하는 예이다.

```
EXEC SQL ENABLE THREADS;
```

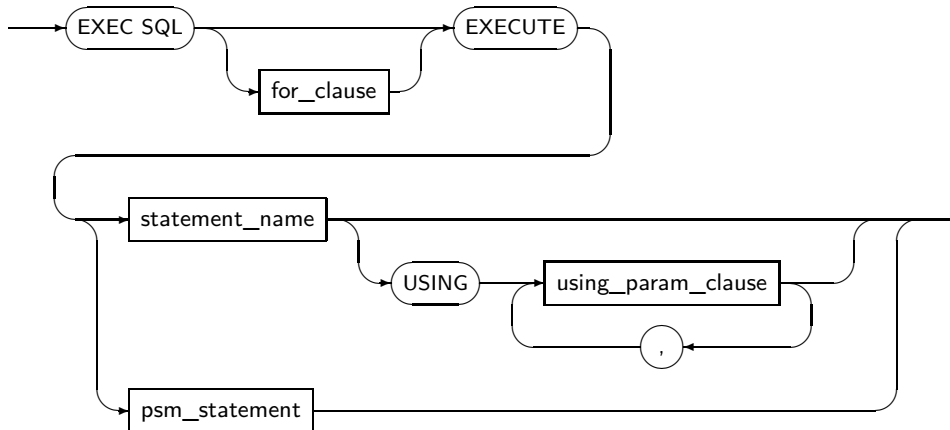
8.4.18. EXECUTE

EXECUTE는 Dynamic SQL 문장을 실행할 때 사용하는 문장이다.

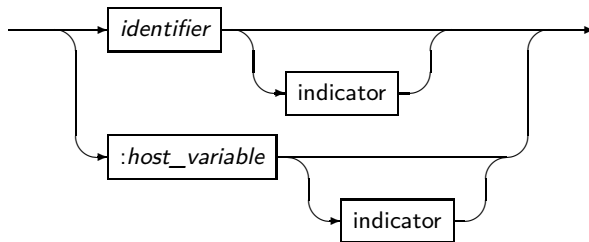
EXECUTE의 세부 내용은 다음과 같다.

- 문법

execute_statement



using_param_clause



- 구성요소

– execute_statement

구성요소	설명
for_clause	for_clause를 사용해 EXECUTE를 수행할 횟수를 지정할 수 있다. 자세한 내용은 “8.3.2. FOR 절”을 참고한다.
statement_name	실행할 문장의 이름을 명시한다. 실행할 문장의 이름은 PREPARE 문장을 통해 이미 준비되어 있어야 한다.
USING using_param_clause	USING 절을 이용하여 입력 변수를 지정한다. 또는 입력 배열 변수를 사용할 수도 있다.
psm_statement	실행할 anonymous psm block 문장의 내용을 서술한다.

– using_param_clause

구성요소	설명
identifier	입력 변수를 정의한 식별자이다.
:host_variable	입력 변수를 명시한다. 입력 변수의 개수는 하나 이상이고, 준비된 문장 내에 포함된 입력 변수의 개수와 같아야 한다.
indicator	지시자 변수를 명시할 때 사용한다.

- 예제

다음은 EXECUTE를 사용하는 예이다.

```
EXEC SQL EXECUTE sql_stmt;

EXEC SQL EXECUTE sql_stmt USING :empno, :deptno;

EXEC SQL FOR :count EXECUTE sql_stmt
      USING :empno_arr INDICATOR :empno_arr_ind;
```

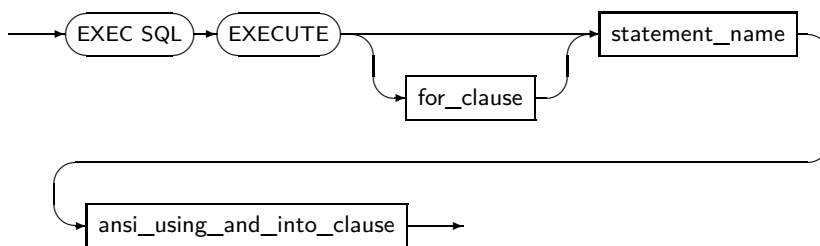
8.4.19. EXECUTE DESCRIPTOR

EXECUTE DESCRIPTOR는 Dynamic SQL 문장을 실행할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

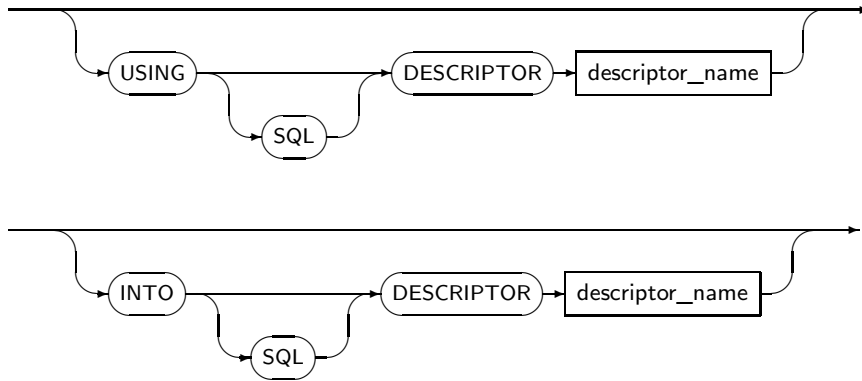
EXECUTE DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

ansi_execute_statement



ansi_using_and_into_clause



- 구성요소

– ansi_execute_statement

구성요소	설명
for_clause	for_clause를 사용해 EXECUTE를 수행할 횟수를 지정할 수 있다. 자세한 내용은 “8.3.2. FOR 절”을 참고한다.
statement_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 서술자 이름을 문자열로 가지고 있어야 한다.

– ansi_using_and_into_clause

구성요소	설명
USING	입력 호스트 변수의 정보를 가지고 있는 서술자를 사용해야 할 경우 명시한다.
INTO	문장 수행 결과가 출력 값을 가지고 있을 경우 사용할 서술자를 명시한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드로 특별한 의미는 없다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

다음은 EXECUTE DESCRIPTOR를 사용하는 예이다.

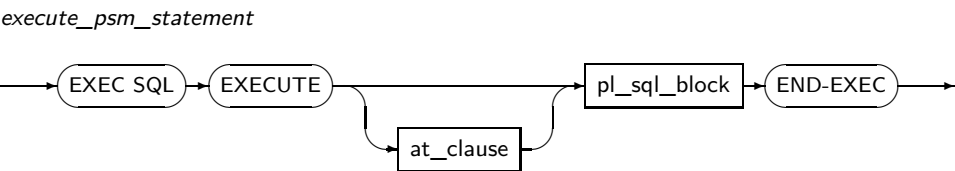
```
EXEC SQL EXECUTE S USING DESCRIPTOR 'input_desc'
      INTO DESCRIPTOR 'output_desc';
```

8.4.20. EXECUTE ... END-EXEC

EXECUTE ... END-EXEC는 PL/SQL 문장을 실행할 때 사용하는 문장이다.

EXECUTE ... END-EXEC의 세부 내용은 다음과 같다.

- 문법



- 구성요소

– *execute_psm_statement*

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
pl_sql_block	사용자가 작성한 PL/SQL 블록을 사용한다. .

- 예제

다음은 EXECUTE ... END-EXEC를 사용하는 예이다.

```
EXEC SQL EXECUTE
  DECLARE
    empnum NUMBER(4);
  BEGIN
    empnum := 1356;
    INSERT INTO EMP ( EMPNUM ) VALUES (empnum);
  END;
END-EXEC;

EXEC SQL EXECUTE
  BEGIN
    UPDATE EMP SET
      EMPNAME = :hv_empname;
      SALARY = :hv_salary;
    WHERE
      EMPNUM = :hv_empnum;
```

```
END;  
END-EXEC;
```

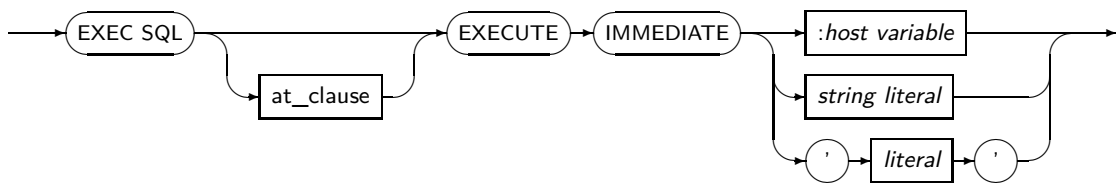
8.4.21. EXECUTE IMMEDIATE

EXECUTE IMMEDIATE는 Dynamic SQL 문장을 준비하지 않고 바로 실행할 때 사용하는 문장이다. 이 문장을 통해 실행할 SQL 문장은 입력 변수가 포함되지 않아야 한다.

EXECUTE IMMEDIATE의 세부 내용은 다음과 같다.

- 문법

execute_immediate_statement



- 구성요소

구성요소	설명
:host_variable	실행할 Dynamic SQL 문장을 호스트 변수를 사용해 명시한다.
string literal	실행할 문장을 문자열 리터럴을 사용해 명시한다.
'literal'	실행할 문장을 문자열을 사용해 명시한다.

- 예제

다음은 EXECUTE IMMEDIATE를 사용하는 예이다.

```
EXEC SQL EXECUTE IMMEDIATE :sql_stmt;  
  
EXEC SQL EXECUTE IMMEDIATE  
  'SELECT EMPNO, ENAME, SALARY FROM EMP';
```

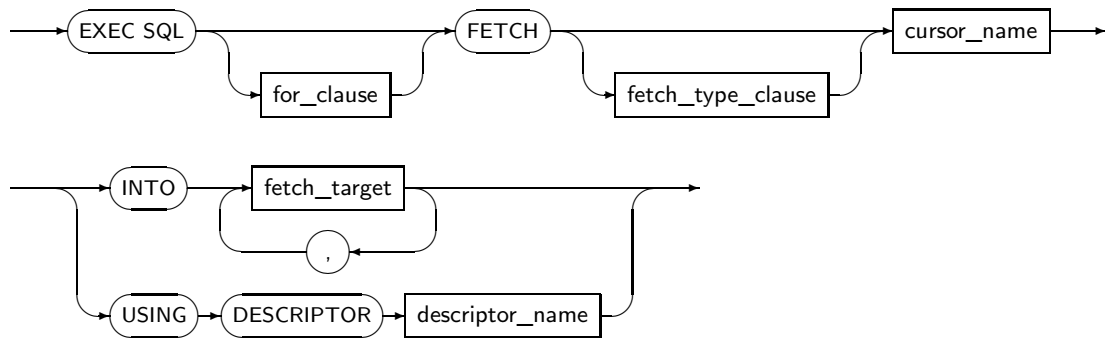
8.4.22. FETCH

FETCH는 커서가 현재 가리키고 있는 로우의 데이터를 읽어 올 때 사용하는 문장이다. 이 문장을 사용할 때 출력 변수로 배열 변수를 사용할 수 있다. 배열 변수를 사용할 경우 여러 개의 로우 데이터를 동시에 읽어 올 수 있다.

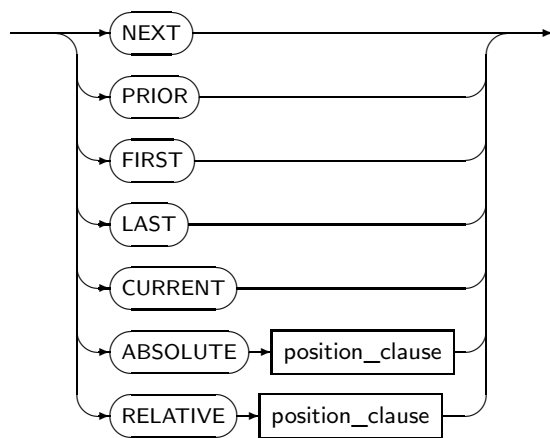
FETCH의 세부 내용은 다음과 같다.

- 문법

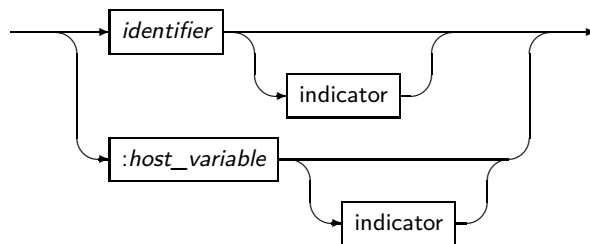
fetch_statement



fetch_type_clause



fetch_target



- 구성요소

– *fetch_statement*

구성요소	설명
<i>for_clause</i>	입력 배열 변수를 사용할 때 <i>for_clause</i>를 사용해 동시에 읽을 로우의 개수를 지정할 수 있다. <i>for_clause</i>가 포함되어 있지 않거나, <i>for_clause</i>에 지정된 로우의 개수가 입력 배열 변수의 크기보다 크면, 배열 변수의 크기만큼 로우를 읽는다. 자세한 내용은 “8.3.2. FOR 절”을 참고한다.

구성요소	설명
fetch_type_clause	스크롤 커서일 경우 스크롤 타입을 명시한다.
cursor_name	커서의 이름을 명시한다. 사용될 커서는 열려있어야 한다.
fetch_target	결과 값을 저장할 호스트 변수를 명시한다.
USING DESCRIPTOR	서술자의 이름을 명시할 경우 서술자의 이름 앞에 붙이는 키워드이다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.

– fetch_type_clause

구성요소	설명
NEXT	현재 커서가 가리키고 있는 로우의 다음 로우에 액세스를 한다. PRIOR 옵션과 반대이다. (생략 가능)
PRIOR	현재 커서가 가리키고 있는 로우의 이전 로우에 액세스를 한다. NEXT 옵션과 반대이다.
FIRST	맨 처음에 위치한 로우에 액세스를 한다. LAST 옵션과 반대이다.
LAST	맨 마지막에 위치한 로우에 액세스를 한다. FIRST 옵션과 반대이다.
CURRENT	현재 로우에 액세스를 한다.
ABSOLUTE position_clause	전체 로우 중에서 position_clause번째 로우에 액세스를 한다.
RELATIVE position_clause	현재 커서가 가리키고 있는 로우의 다음 position_clause번째에 위치한 로우에 액세스를 한다. position_clause의 값이 음수라면 커서가 현재 위치에서 앞으로 이동한다. 예를 들어 현재 커서가 8번째 로우를 가리키고 있는데, 'FETCH RELATIVE -3'을 실행한다면 커서는 5번째 로우를 가리키게 된다.

– fetch_target

구성요소	설명
identifier	출력 변수를 정의한 식별자이다.
:host_variable	출력 변수를 명시한다. 출력 변수의 개수는 하나 이상이고, 준비된 문장의 결과로 출력될 SELECT 리스트의 컬럼 개수와 동일해야 한다.
indicator	지시자 변수를 명시할 때 사용한다.

● 예제

다음은 FETCH를 사용하는 예이다.

```
EXEC SQL FETCH emp_cursor INTO :empno, :ename, :salary;

EXEC SQL FETCH emp_cursor
      INTO :empno, :ename:ename_ind, :salary:salary_ind;

EXEC SQL FOR :count FETCH emp_cursor
      INTO :empno_arr, :ename_arr, :salary_arr;
```

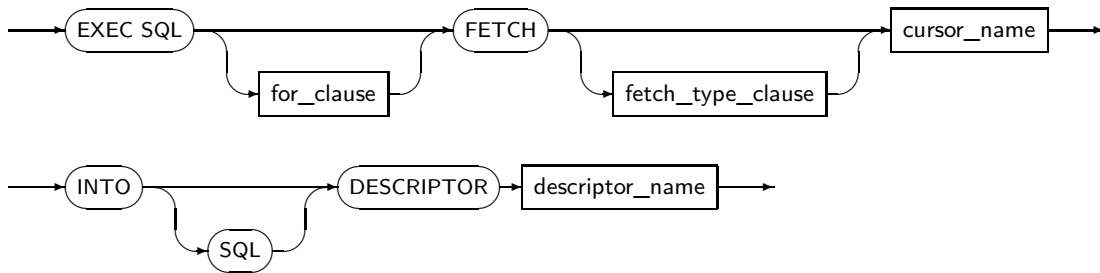
8.4.23. FETCH DESCRIPTOR

FETCH DESCRIPTOR는 커서가 현재 가리키고 있는 로우 데이터를 읽어 올 때 사용하는 문장이다. FETCH와 거의 동일하게 동작한다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

FETCH DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

ansi_fetch_statement



- 구성요소

구성요소	설명
for_clause	입력 배열 변수를 사용할 때 for_clause를 사용해 동시에 읽을 로우의 개수를 지정할 수 있다. for_clause가 포함되어 있지 않거나, for_clause에 지정된 로우의 개수가 입력 배열 변수의 크기보다 크면, 배열 변수의 크기만큼 로우를 읽는다. 자세한 내용은 “8.3.2. FOR 절”을 참고한다.
fetch_type_clause	FETCH의 fetch_type_clause와 동일하다.
cursor_name	커서의 이름을 명시한다. 사용될 커서는 열려있는 커서이어야 한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드로 특별한 의미는 없다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다.

구성요소	설명
	호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

다음은 FETCH DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL FETCH C1 INTO DESCRIPTOR 'output_descriptor';
```

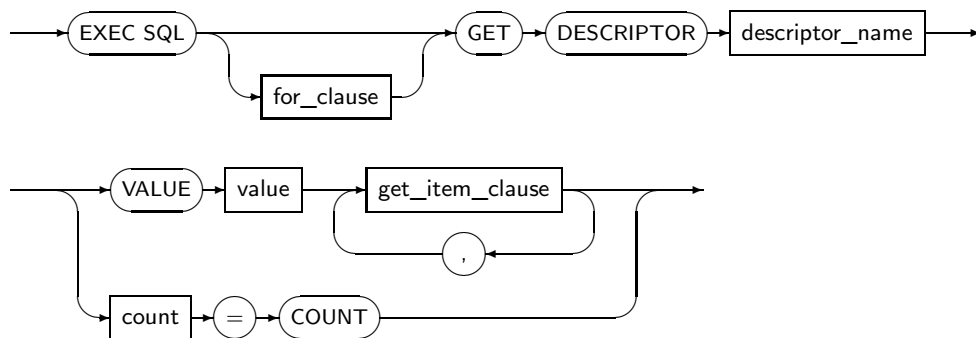
8.4.24. GET DESCRIPTOR

GET DESCRIPTOR는 지정한 서술자에서 원하는 정보를 가져올 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

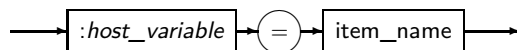
GET DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

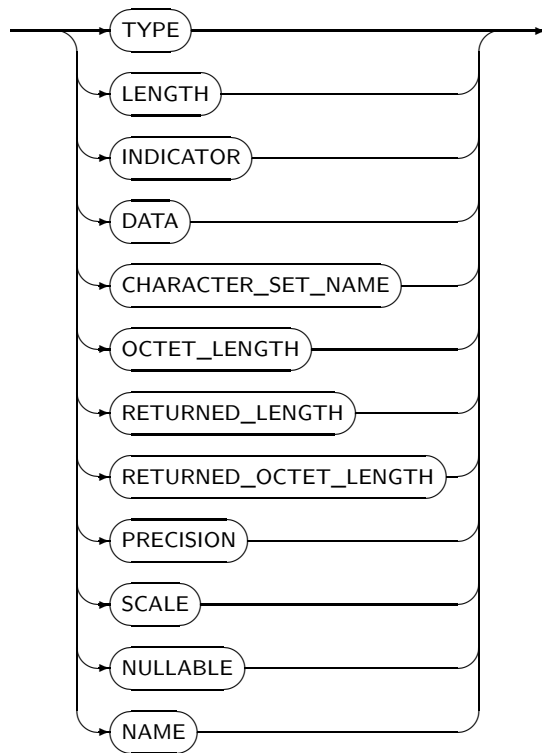
get_descriptor_statement



get_item_clause



item_name



- 구성요소

- get_descriptor_statement

구성요소	설명
for_clause	입력 배열 변수를 사용할 때 for_clause를 사용해 동시에 읽을 로우의 개수를 지정할 수 있다. for_clause가 포함되어 있지 않거나, for_clause에 지정된 로우의 개수가 입력 배열 변수의 크기보다 크면, 배열 변수의 크기만큼 로우를 읽는다. 자세한 내용은 “8.3.2. FOR 절”을 참고한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.
VALUE value	정보를 가져올 호스트 변수의 순서를 지정한다.
get_item_clause	가져올 정보의 항목과 정보를 저장할 호스트 변수를 명시한다.
count	사용된 호스트 변수의 개수를 알고자 할 경우 그 개수를 저장할 호스트 변수를 명시한다.

– value

구성요소	설명
:host_variable	값이 들어 있는 호스트 변수를 사용할 때 명시한다.
integer	값을 직접 정수로 입력할 때 사용한다.

– get_item_clause

구성요소	설명
:host_variable	해당 항목을 가져올 호스트 변수를 명시한다.
item_name	값을 가져올 구체적인 항목을 명시한다.

– item_name

구성요소	설명
TYPE	데이터의 타입을 가져오고자 할 때 사용한다.
LENGTH	데이터의 최대 길이를 가져오고자 할 때 사용한다.
INDICATOR	데이터 연관된 지시자 값을 가져오고자 할 때 사용한다.
DATA	데이터 값을 가져오고자 할 때 사용한다.
CHARACTER_SET_NAME	데이터가 저장된 컬럼의 문자 세트를 가져오고자 할 때 사용한다.
OCTET_LENGTH	데이터 길이를 Byte 단위로 환산해 가져오고자 할 때 사용한다.
RETURNED_LENGTH	FETCH를 할 때 실제로 받아들일 데이터 길이를 가져오고자 할 때 사용한다.
RETURNED_OCTET_LENGTH	FETCH를 할 때 실제로 받아들일 데이터 길이를 Byte 단위로 환산해 가져오고자 할 때 사용한다.
PRECISION	받아올 데이터의 정밀도를 가져오고자 할 때 사용한다.
SCALE	받아올 데이터의 스케일을 가져오고자 할 때 사용한다.
NULLABLE	해당 컬럼의 데이터가 NULL이 될 수 있는지 여부를 알고자 할 때 사용한다. – 이 값이 1이면, 해당 컬럼은 NULL값을 가질 수 있다. – 이 값이 0이면, 해당 컬럼은 NULL값을 가질 수 없는 키이거나 NOT NULL 제약조건을 가지고 있는 컬럼이다.
NAME	해당 컬럼의 이름을 가져오고자 할 때 사용한다.

● 예제

다음은 GET DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL GET DESCRIPTOR 'input_descriptor' :input_count = COUNT;
```

8.4.25. INSERT

INSERT는 테이블 또는 뷰에 로우를 삽입할 때 사용하는 문장이다. 전체 컬럼 또는 일부 컬럼에 데이터를 삽입할 수 있다. **INSERT**를 사용할 때, 입력 변수로 배열 변수를 사용할 수도 있으며, 지시자 변수를 함께 사용할 수 있다. 또한 입력할 데이터의 값을 사용자가 직접 지정할 수도 있고, 부질의를 통하여 지정할 수도 있다. 부질의를 이용하는 경우 부질의의 결과 로우 모두가 테이블이나 뷰에 삽입된다.

INSERT의 세부 내용은 다음과 같다.

- 문법

INSERT의 문법에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

- 특권

INSERT를 사용하려면, **INSERT**의 대상 테이블 또는 뷰에 대하여 **INSERT** 객체 특권을 갖거나 **INSERT ANY TABLE** 시스템 특권을 갖고 있어야 한다.

- 구성요소

구성요소	설명
FOR	입력 배열 변수와 함께 사용될 경우 FOR 절을 이용하여 삽입할 데이터의 개수를 지정할 수 있다. FOR 절이 포함되어 있지 않거나 FOR 절에서 지정한 크기가 입력 배열 변수의 크기보다 크면, 전체 배열 변수에 저장된 데이터가 테이블에 삽입된다.

- 예제

다음은 **INSERT**를 사용하는 예이다.

```
EXEC SQL INSERT INTO EMP VALUES (34, 'James', 45000, 'NewYork');

EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
VALUES (:empno, :ename);

EXEC SQL FOR :count INSERT INTO EMP (EMPNO, ENAME)
VALUES (:empno_arr, :ename_arr :ename_arr_ind);

EXEC SQL INSERT INTO HIGH_EMP
SELECT * FROM EMP WHERE SALARY > 50000;
```

위의 예에서 알 수 있듯이 **INSERT** 문장은 **EXEC SQL**로 시작한다는 것을 제외하면, 일반적인 **SQL** 문장의 문법과 크게 다르지 않다.

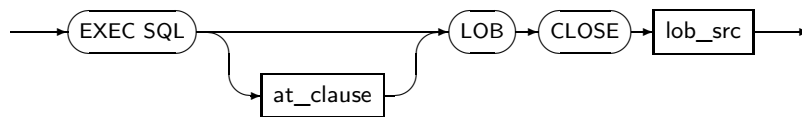
8.4.26. LOB CLOSE

LOB CLOSE는 OPEN 된 LOB 사용이 끝난 후 닫아주기 위해 사용하는 문장이다. 이미 닫힌 LOB에 대해 또 다시 닫으려 할 경우 에러가 발생하고, 또한 사용자에게 의해 닫히기 전에 COMMIT 이 발생할 경우에도 에러가 발생한다.

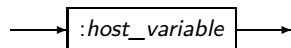
LOB CLOSE의 세부 내용은 다음과 같다.

- 문법

lob_close_statement



lob_src



- 구성요소

– lob_close_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절” 을 참고한다.
lob_src	대상이 되는 대용량 객체형의 데이터의 LOB 지시자 변수이다.

– lob_src

구성요소	설명
:host_variable	사용할 LOB 지시자 변수를 명시한다.

- 예제

다음은 LOB CLOSE를 사용하는 예이다.

```
EXEC SQL LOB CLOSE :lob_loc;
```

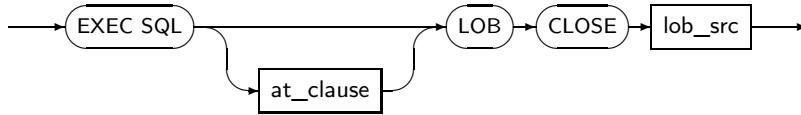
8.4.27. LOB CREATE TEMPORARY

LOB CREATE TEMPORARY는 temporary LOB을 생성하기 위해 사용하는 문장이다.

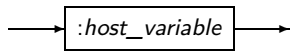
LOB CREATE TEMPORARY의 세부 내용은 다음과 같다.

- 문법

lob_close_statement



lob_src



- 구성요소

– lob_create_temp_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
lob_src	대상이 되는 대용량 객체형 데이터의 LOB 지시자 변수이다.

– lob_src

구성요소	설명
:host_variable	사용할 LOB 지시자 변수를 명시한다.

- 예제

다음은 LOB CREATE TEMPORARY를 사용하는 예이다.

```
EXEC SQL LOB CREATE TEMPORARY :lob_loc;
```

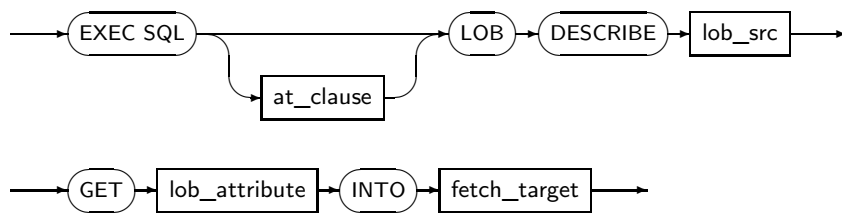
8.4.28. LOB DESCRIBE

LOB DESCRIBE는 대용량 객체형의 데이터 타입의 속성을 받아올 때 사용하는 문장이다. 이 문장은 LOB READ를 실행하기 전 버퍼를 마련하기 위해서 대용량 객체형의 데이터의 길이를 알아내는 등의 용도로 사용한다.

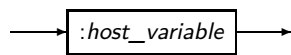
LOB DESCRIBE의 세부 내용은 다음과 같다.

- 문법

lob_describe_statement



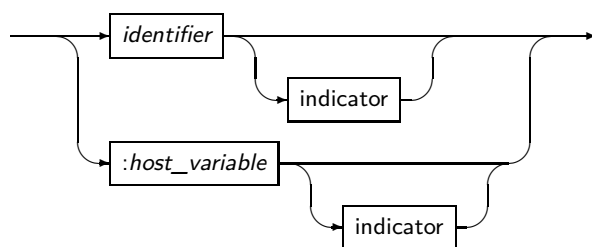
lob_src



lob_attribute



fetch_target



- 구성요소

– lob_describe_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
lob_src	속성을 받아올 대상이 되는 대용량 객체형의 데이터의 LOB 지시자 변수이다.
lob_attribute	가져올 대용량 객체형의 데이터 타입의 속성을 지정한다.
fetch_target	대용량 객체형의 데이터 타입의 속성이 저장될 변수이다.

– lob_src

구성요소	설명
::host_variable	사용할 LOB 지시자 변수를 명시한다.

– lob_attribute

구성요소	설명
LENGTH	LOB의 길이를 알고자 할 때 사용한다.

– fetch_target

구성요소	설명
identifier	길이 값을 받아올 호스트 변수를 콜론(:) 없이 사용할 수 있다.
:host_variable	길이 값을 받아올 호스트 변수를 명시한다.
indicator	지시자 변수를 명시할 때 사용한다.

● 예제

다음은 LOB DESCRIBE를 사용하는 예이다.

```
ClobLocator *clob;
unsigned int clob_len;

EXEC SQL ALLOCATE :clob;

EXEC SQL SELECT clob_col INTO :clob FROM CLOB_TABLE WHERE num_col = 1;

EXEC SQL LOB DESCRIBE :clob GET LENGTH INTO :clob_len;
```

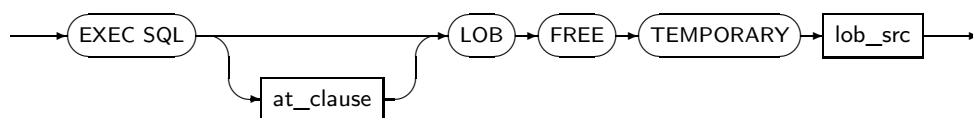
8.4.29. LOB FREE TEMPORARY

LOB FREE TEMPORARY는 생성된 temporary LOB에 할당된 메모리를 해제하기 위해 사용하는 문장이다.

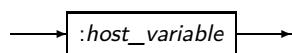
LOB FREE TEMPORARY의 세부 내용은 다음과 같다.

● 문법

lob_free_temp_statement



lob_src



● 구성요소

– lob_free_temp_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다.

구성요소	설명
	데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
lob_src	대상이 되는 대용량 객체형 데이터의 LOB 지시자 변수이다.

– lob_src

구성요소	설명
:host_variable	사용할 LOB 지시자 변수를 명시한다.

- 예제

다음은 LOB FREE TEMPORARY를 사용하는 예이다.

```
EXEC SQL LOB FREE TEMPORARY :lob_loc;
```

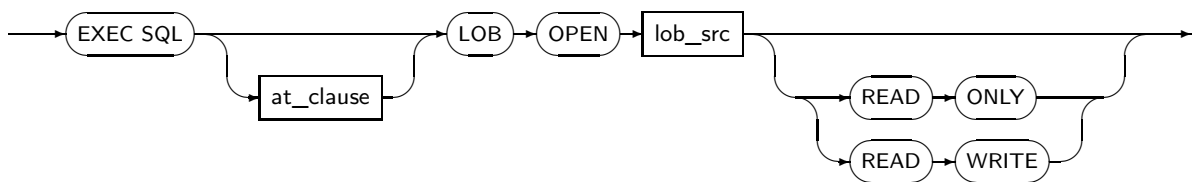
8.4.30. LOB OPEN

LOB OPEN은 LOB이나 BFILE을 열기 위해 사용하는 문장이다. LOB OPEN 문장 이후에 READ ONLY를 통해 읽기 모드를, WRITE ONLY를 통해 쓰기 모드를 사용할 수 있다. 지정하지 않으면 기본값으로 2가지 모드를 모두 사용할 수 있다.

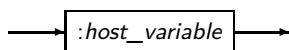
LOB OPEN의 세부 내용은 다음과 같다.

- 문법

lob_open_statement



lob_src



- 구성요소

– lob_open_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다.

구성요소	설명
	데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
lob_src	속성을 받아올 대상이 되는 대용량 객체형의 데이터의 LOB 지시자 변수이다.

– lob_src

구성요소	설명
::host_variable	사용할 LOB 지시자 변수를 명시한다.

- 예제

다음은 LOB OPEN를 사용하는 예이다.

```
EXEC SQL LOB OPEN :lob_src;
```

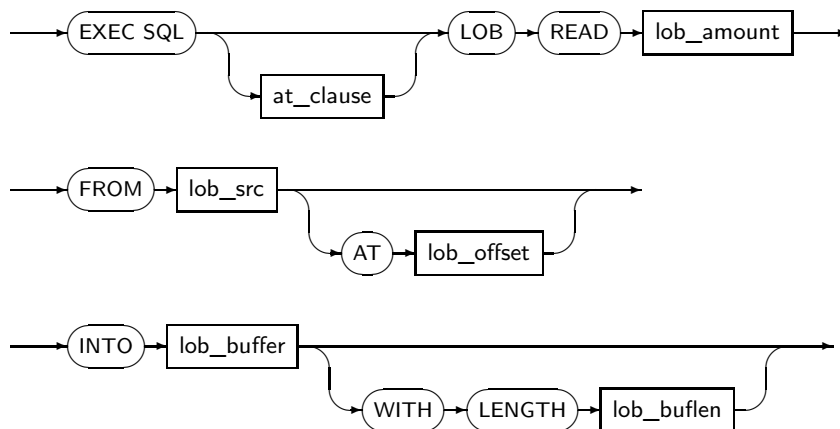
8.4.31. LOB READ

LOB READ는 LOB 지시자가 가리키고 있는 위치의 대용량 객체형의 데이터의 내용을 읽어 올 때 사용하는 문장이다.

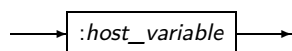
LOB READ의 세부 내용은 다음과 같다.

- 문법

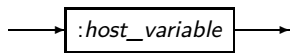
lob_read_statement



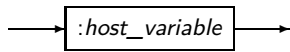
lob_amount



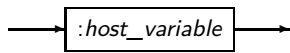
lob_offset



lob_buffer



lob_bufllen



- 구성요소

- lob_read_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
lob_amount	읽어올 대용량 객체형의 데이터의 길이를 명시한다. 데이터를 읽어 온 후에는 실제로 읽어 온 길이가 기록된다. (0~2GB)
lob_src	속성을 받아올 대상이 되는 대용량 객체형의 데이터의 LOB 지시자 변수이다.
lob_offset	대용량 객체형의 데이터를 읽기 시작할 위치를 명시한다.
lob_buffer	대용량 객체형의 데이터를 읽어 올 때 사용할 버퍼의 이름을 명시한다. 버퍼로 사용될 메모리는 미리 할당되어 있어야 한다.
lob_bufllen	대용량 객체형의 데이터를 읽어 올 버퍼의 길이를 명시한다. 이곳에 명시된 길이만큼만 읽는다. (0~2GB)

- lob_amount

구성요소	설명
:host_variable	대용량 객체형의 데이터의 길이를 값으로 갖는 호스트 변수를 명시한다.

- lob_offset

구성요소	설명
:host_variable	대용량 객체형의 데이터를 읽기 시작할 위치를 값으로 갖는 호스트 변수를 명시한다.

- lob_buffer

구성요소	설명
:host_variable	대용량 객체형의 데이터를 읽어 올 때 사용할 버퍼의 이름을 값으로 갖는 호스트 변수를 명시한다.

– lob_bufllen

구성요소	설명
:host_variable	대용량 객체형의 데이터를 읽어 올 때 사용할 버퍼의 길이를 값으로 갖는 호스트 변수를 명시한다.

- 예제

다음은 LOB READ를 사용하는 예이다.

```
ClobLocator *clob;
unsigned int read_amt;
unsigned char buffer[26];

EXEC SQL ALLOCATE :clob;

EXEC SQL SELECT clob_col INTO :clob FROM CLOB_TABLE WHERE num_col = 1;

read_amt = 25;

EXEC SQL LOB READ :read_amt FROM :clob
      INTO :buffer WITH LENGTH :read_amt;
```

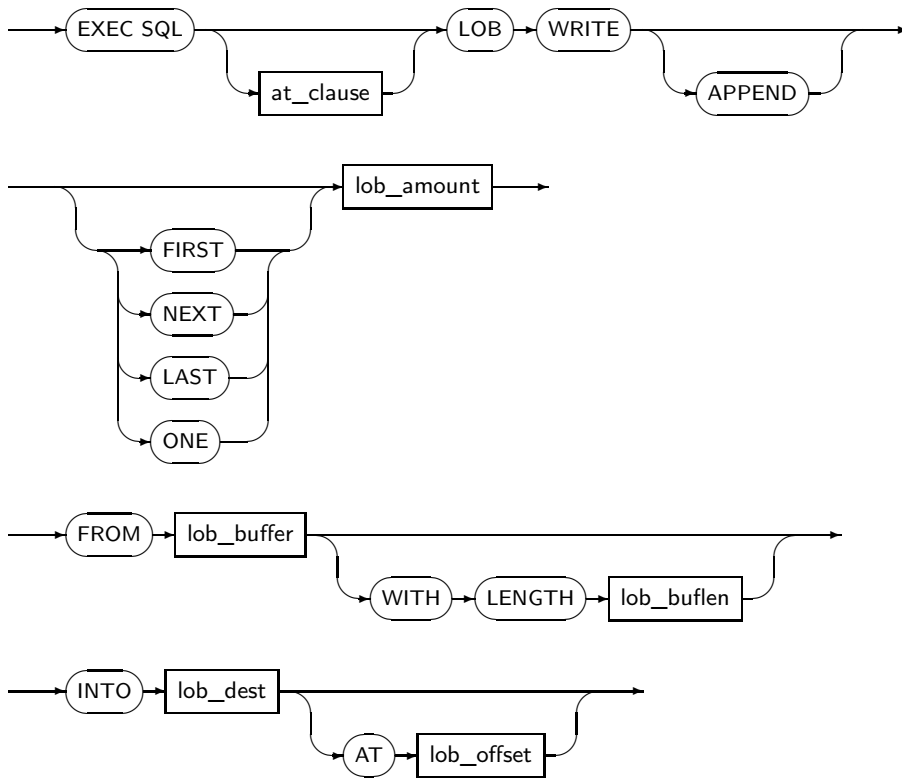
8.4.32. LOB WRITE

LOB WRITE는 LOB 지시자가 가리키는 위치에 명시된 해당 내용에 쓰기를 실행할 때 사용하는 문장이다.

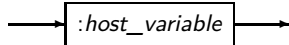
LOB WRITE의 세부 내용은 다음과 같다.

- 문법

lob_write_statement



lob_dest



- 구성요소

– lob_write_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절”을 참고한다.
APPEND	쓰고자 하는 내용이 이미 있는 LOB 데이터의 가장 마지막에 더해진다. 이 경우에 별도의 offset 이 입력될 경우 에러가 발생한다.
FIRST	대용량 객체형 데이터를 여러 번으로 나누어서 쓸 경우 맨 처음에 해당되는 조각을 쓰려고 할 때 명시한다.
NEXT	대용량 객체형 데이터를 여러 번으로 나누어서 쓸 경우 쓰기 시작한 LOB 에 계속해서 쓰려고 할 때 명시한다.
LAST	대용량 객체형 데이터를 여러 번으로 나누어서 쓸 경우 마지막으로 쓸 내용일 경우 명시한다.

구성요소	설명
ONE	입력 내용을 나누어 쓰지 않고 버퍼에 있는 내용이 쓰려고 하는 전체 내용일 경우이다. 쓰기 옵션 중 어떤 내용도 지정하지 않을 경우 이 값이 기본값이다.
lob_amount	대용량 객체형의 데이터의 길이를 명시한다. 데이터 쓰기가 실행된 후에는, 실제로 쓰기가 실행된 데이터의 길이가 기록된다. (0~2GB)
lob_buffer	쓰기가 실행될 해당 내용이 들어 있는 버퍼를 명시한다.
lob_buflen	쓰기가 실행될 해당 내용이 들어 있는 버퍼의 길이를 명시한다. (0~2GB)
lob_dest	쓰기가 실행될 LOB 지시자를 명시한다.
lob_offset	쓰기가 실행될 시작 위치를 명시한다.

– lob_dest

구성요소	설명
:host_variable	LOB 지시자가 저장된 변수를 명시한다.

- 예제

다음은 LOB WRITE를 사용하는 예이다.

```

BlobLocator *blob;
unsigned int blob_len, offset;
unsigned char buffer[MAXBUFLen];
FILE *fp;

EXEC SQL ALLOCATE :blob;

EXEC SQL INSERT INTO BLOB_TABLE VALUES (empty_blob())
RETURNING blob_col INTO :blob;

fp = fopen((const char *)"test_blob.jpeg", (const char *)"r");
fseek(fp, 0L, SEEK_END);
blob_len = (unsigned int)ftell(fp);

fseek(fp, 0L, SEEK_SET);
write_amt = MAXBUFLen;
fread((void *)buffer, (size_t)write_amt, (size_t)1, fp);
offset = 1;

EXEC SQL LOB WRITE FIRST :write_amt FROM :buffer
WITH LENGTH :write_amt INTO :blob AT :offset;

```

8.4.33. OPEN

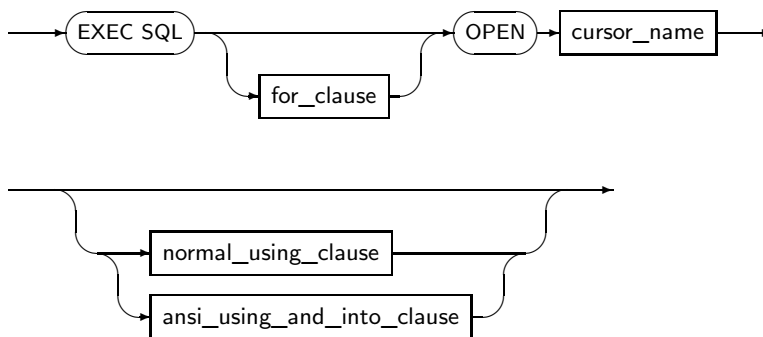
OPEN은 커서를 열 때 사용하는 문장이다. 이 문장을 사용해 열려고 하는 커서는 이미 선언되어 있어야 하며, **SELECT** 문장과 연관되어 있어야 한다. 커서의 선언은 **DECLARE CURSOR**를 통하여 실행된다.

커서를 여는 것과 동시에 연관된 **SELECT** 문장이 실행되며, 커서는 질의 문장이 반환한 결과의 제일 처음에 위치한 로우를 가리킨다. 커서는 **SELECT** 문장과 연관되기도 하지만 **Dynamic SQL** 문장과 연관되기도 한다.

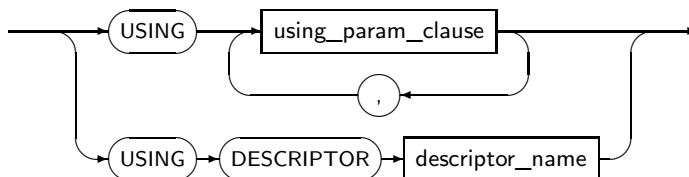
OPEN의 세부 내용은 다음과 같다.

- 문법

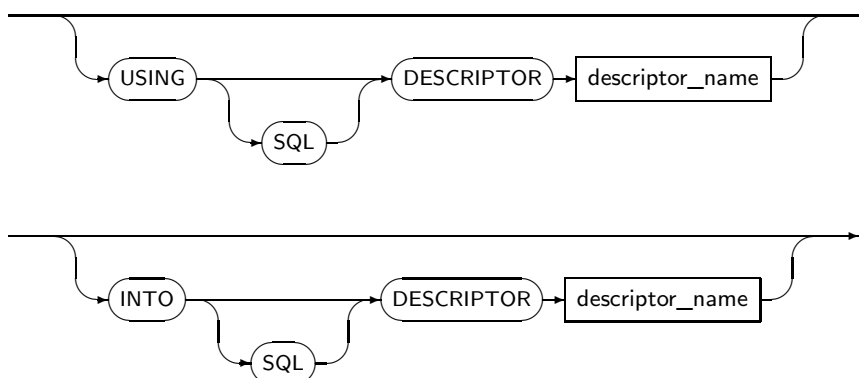
open_statement



normal_using_clause



ansi_using_and_into_clause



- 구성요소

- open_statement

구성요소	설명
for_clause	실행 반복 횟수를 지정한다. 자세한 내용은 “8.3.2. FOR 절”을 참고한다.
cursor_name	열려고 하는 커서의 이름을 명시한다.
normal_using_clause	일반적으로 사용하는 OPEN 문장의 형태이다.
ansi_using_and_into_clause	이미 정의된 서술자를 이용한다. ANSI 타입의 Dynamic SQL 문장에만 사용된다.

– normal_using_clause

구성요소	설명
USING	커서가 Dynamic SQL 문장과 연관된 경우 USING 절을 이용하여 SELECT 문장의 입력 변수에 할당할 값을 지정할 수 있다. 입력 변수로 배열 변수를 사용할 수도 있으며, 서술자 변수를 함께 사용할 수도 있다.
using_param_clause	입력 호스트 변수의 정보가 필요할 경우에 사용한다.
USING DESCRIPTOR	입력 호스트 변수의 정보를 가져왔던 서술자를 사용할 경우 명시한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.

– ansi_using_and_into_clause

구성요소	설명
USING	입력 호스트 변수의 정보가 저장된 서술자 변수가 필요할 경우에 사용한다.
INTO	출력 호스트 변수의 정보가 저장된 서술자 변수가 필요할 경우에 사용한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드로 특별한 의미는 없다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름”을 참고한다.

● 예제

다음은 OPEN을 사용하는 예이다.

```
EXEC SQL OPEN emp_cursor;

EXEC SQL OPEN emp_cursor USING :empno;

EXEC SQL FOR :count OPEN emp_cursor
      USING :empno INDICATOR :empno_ind;
```

8.4.34. PREPARE

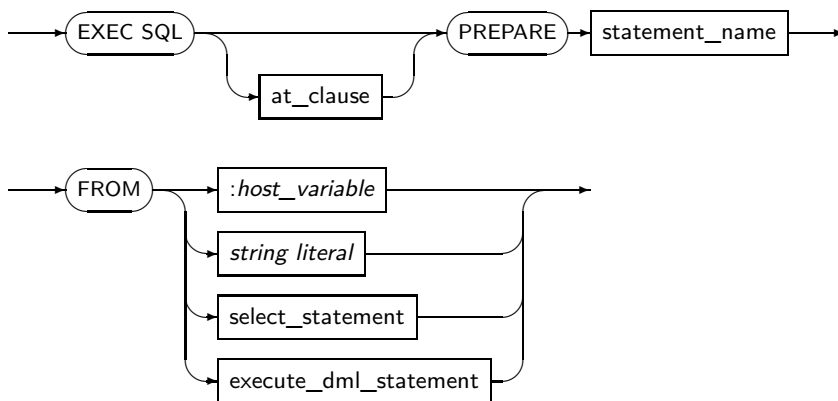
PREPARE는 Dynamic SQL 문장을 준비할 때 사용하는 문장이다. 준비된 SQL 문장은 문장의 이름을 통해 서로 식별되며, 이후에 **DECLARE CURSOR** 또는 **EXECUTE**에서 참조된다.

SQL 문장을 준비한다는 것은 단지 문장을 파싱(Parsing)한다는 의미일 뿐이며, 문장이 실행된다는 의미는 아니다. 문장이 실제로 실행되는 것은 **EXECUTE** 문장을 통해서이다. SQL 문장에는 하나 이상의 입력 변수가 포함될 수 있는데, 입력 변수가 포함될 때 입력 변수의 이름은 별다른 의미를 갖지 않으며, **tbESQL** 프로그램에 미리 선언되어 있을 필요도 없다.

PREPARE의 세부 내용은 다음과 같다.

- 문법

prepare_statement



- 구성요소

구성요소	설명
statement_name	준비를 실행할 SQL 문장의 이름을 명시한다.
FROM	FROM 절 뒤에는 준비할 Dynamic SQL 문장이 온다. SQL 문장 자체가 올 수도 있으며, SQL 문장의 문자열을 저장한 호스트 변수가 올 수도 있다. SELECT 문장이라면 문장 자체가 올 수 있다.
:host_variable	Dynamic SQL 문장을 저장한 호스트 변수를 명시한다.

구성요소	설명
string literal	Dynamic SQL 문장을 나타내는 문자열을 명시한다.
select_statement	SELECT 문장 자체를 명시한다.
execute_dml_statement	실행할 INSERT, UPDATE, DELETE 문장 자체를 명시한다.

- 예제

다음은 PREPARE를 사용하는 예이다.

```
EXEC SQL PREPARE emp_stmt FROM :sql_stmt;

EXEC SQL PREPARE emp_stmt
      FROM 'UPDATE EMP SET SALARY = SALARY * 1.05';

EXEC SQL PREPARE emp_stmt FROM
      SELECT EMPNO, ENAME, SALARY FROM EMP WHERE DEPTNO = :deptno;
```

8.4.35. ROLLBACK

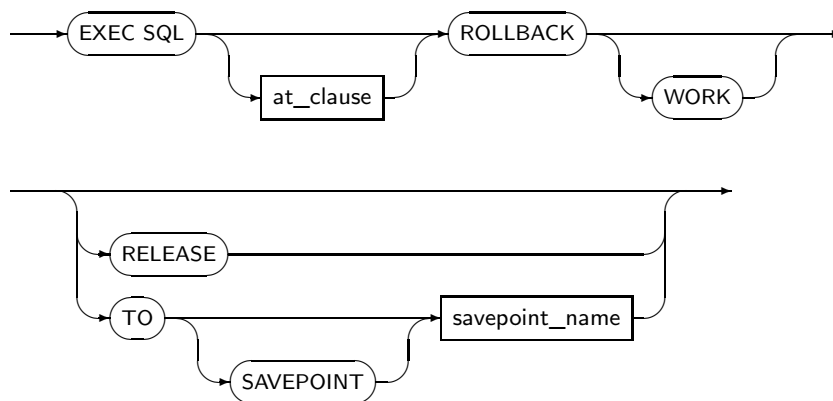
ROLLBACK은 현재 진행 중인 트랜잭션을 롤백하고 갱신된 모든 내용을 취소할 때 사용하는 문장이다.

롤백을 실행하는 것과 동시에 데이터베이스와의 접속을 끊을 수도 있으며, 미리 설정된 저장점까지 부분 롤백(Partial Rollback)을 수행할 수도 있다.

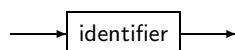
ROLLBACK의 세부 내용은 다음과 같다.

- 문법

rollback_statement



savepoint_name



- 구성요소

- rollback_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “8.3.1. AT 절” 을 참고한다.
WORK	기존 ESQL 프로그램과의 호환성을 위한 키워드로 특별한 의미는 없다.
RELEASE	모든 리소스를 반환하고 데이터베이스와의 접속을 종료한다.
TO	특정 저장점까지 부분 롤백을 수행하고자 할 때 명시한다.
SAVEPOINT	단지 문법 호환을 위해 지원하는 부분이다. 이 부분의 명시여부는 문장 실행에 어떤 영향도 없다.
savepoint_name	부분 롤백을 수행할 저장점의 이름을 명시한다. 저장점은 롤백을 실행하기 전에 프로그램 내에 이미 설정되어 있어야 한다.

- savepoint_name

구성요소	설명
identifier	저장점의 이름을 정의한 식별자이다.

- 예제

다음은 ROLLBACK을 사용하는 예이다.

```
EXEC SQL ROLLBACK;

EXEC SQL ROLLBACK TO SAVEPOINT sp1;

EXEC SQL ROLLBACK WORK RELEASE;
```

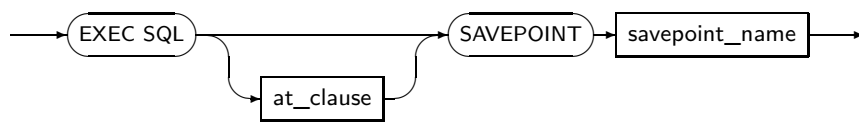
8.4.36. SAVEPOINT

SAVEPOINT는 저장점을 설정할 때 사용하는 문장이다. 설정된 저장점은 특정 지점까지 부분 롤백을 하고자 할 때 사용된다.

SAVEPOINT의 세부 내용은 다음과 같다.

- 문법

savepoint_statement



- 구성요소

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 "8.3.1. AT 절" 을 참고한다.
savepoint_name	설정할 저장점에 부여할 이름을 명시한다.

- 예제

다음은 SAVEPOINT를 사용하는 예이다.

```
EXEC SQL SAVEPOINT sp1;
```

8.4.37. SELECT

SELECT는 테이블 또는 뷰에 대해 질의를 수행하고, 질의를 수행한 결과 로우의 각 데이터를 출력 변수에 저장할 때 사용하는 문장이다.

SELECT의 세부 내용은 다음과 같다.

- 문법

SELECT의 문법에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

- 특권

SELECT 문장을 사용해 질의를 수행하기 위해서는 대상 테이블에 대한 **SELECT** 객체 특권이 있거나 **SELECT ANY TABLE** 시스템 권한을 갖고 있어야 한다.

- 구성요소

구성요소	설명
INTO	질의 결과의 컬럼의 개수는 INTO 절에 포함된 출력 변수의 개수와 같아야 한다. INTO 절에는 지시자 변수와 함께 출력 배열 변수를 사용할 수 있다. 출력 변수는 모두 스칼라 변수이거나 또는 모두 배열 변수이어야 하며, 이 두 가지가 서로 섞여 있을 수 없다.

구성요소	설명
WHERE	WHERE 절에는 입력 변수를 포함할 수 있다. 이때 지시자 변수와 함께 사용될 수 있다. SELECT 문장 내에서는 입력 배열 변수를 사용할 수 없다.
HAVING	HAVING 절에는 입력 변수를 포함할 수 있다. 지시자 변수 및 입력 배열 변수와 관련된 내용은 WHERE 절과 동일하다.

- 예제

다음은 SELECT를 사용하는 예이다.

```
EXEC SQL SELECT EMPNO, ENAME, SALARY
      INTO :empno, :ename, :salary FROM EMP;

EXEC SQL SELECT EMPNO, ENAME, DNAME
      INTO :empno_arr, :ename_arr :ename_arr_ind, :dname_arr
      FROM EMP E, DEPT D
      WHERE E.DEPTNO = D.DEPTNO AND E.DEPTNO = :deptno;
```

위의 예에서 알 수 있듯이 SELECT 문장은 INTO 절을 제외하면, 일반적인 SQL 문장의 문법과 크게 다르지 않다.

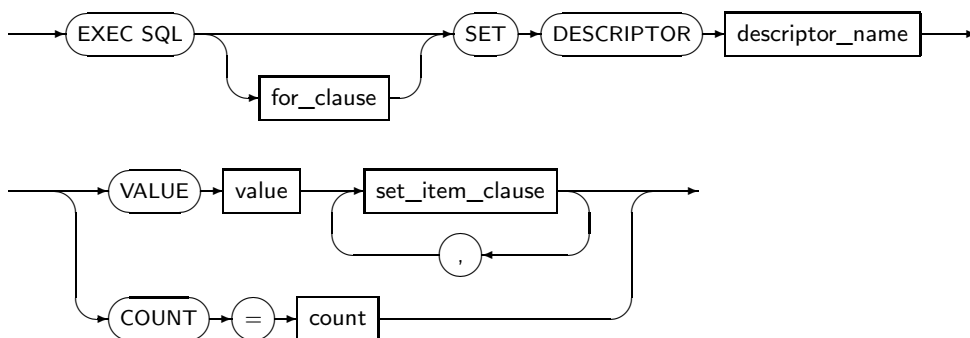
8.4.38. SET DESCRIPTOR

SET DESCRIPTOR는 사용자가 입력해야 할 정보를 지정한 서술자에 기록하는 문장이다. 사용자는 이 문장을 통해 데이터 값이나 데이터 값의 길이 등을 지정할 수 있다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

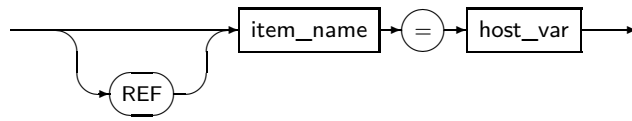
SET DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

set_descriptor_statement



set_item_clause



- 구성요소

- set_descriptor_statement

구성요소	설명
for_clause	반복 횟수를 지정한다. 자세한 내용은 “8.3.2. FOR 절” 을 참고한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “8.3.3. DESCRIPTOR 이름” 을 참고한다.
VALUE value	몇 번째 출력 호스트 변수인지 명시한다.
set_item_clause	출력 호스트 변수의 정보를 입력하기 위해 서술한다.
COUNT	출력 호스트 변수의 개수를 받아오기 위해 서술한다.
count	출력 호스트 변수의 개수를 받아올 호스트 변수를 명시한다.

- set_item_clause

구성요소	설명
REF	INDICATOR, DATA, RETURNED_LENGTH 항목을 지정할 때만 사용할 수 있는 키워드로, 속도와 편리성을 위해 사용한다. 호스트 변수의 값이 아닌 호스트 변수 자체를 지정하며, GET DESCRIPTOR 문장을 따로 실행하지 않아도 FETCH한 후 지정된 항목의 값이 지정된 호스트 변수에 들어간다.
item_name	GET DESCRIPTOR와 동일하다.
host_var	출력 호스트 변수의 각 정보를 받아올 호스트 변수를 지정한다.

- 예제

다음은 SET DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL SET DESCRIPTOR 'output_descriptor' VALUE :occurs
      TYPE = :type, LENGTH = :len;
```

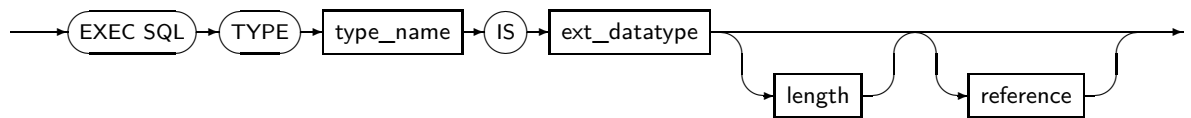
8.4.39. TYPE

TYPE은 사용자가 지정한 데이터 타입을 외부 데이터 타입으로 지정할 때 사용하는 문장이다. **TYPE** 문장을 통해서 호스트 변수의 타입을 사용자가 지정한 타입으로 변경하여 사용할 수 있다. **TYPE** 문장을 사용하기 위해선 사용자가 지정한 데이터 타입이 **TYPE** 문장 전에 반드시 선언되어 있어야 한다.

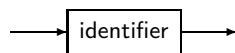
TYPE의 세부 내용은 다음과 같다.

- 문법

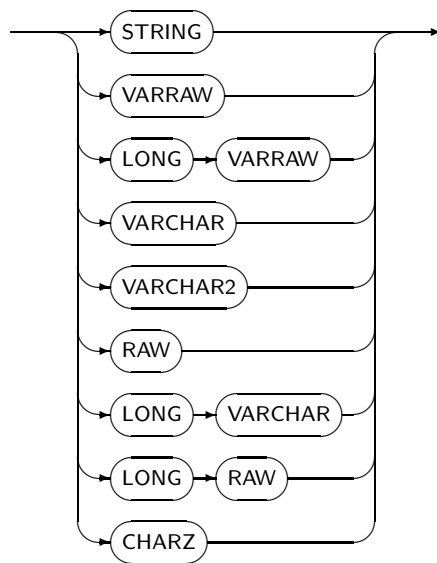
type_statement



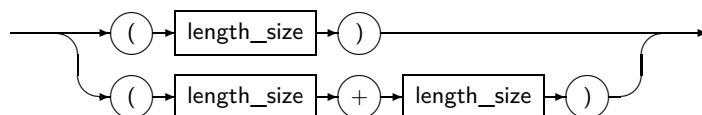
type_name



ext_datatype



length



reference



- 구성요소

– *type_statement*

구성요소	설명
type_name	사용자가 지정한 데이터 타입의 이름을 명시한다.
ext_datatype	사용자가 지정할 데이터 타입을 명시한다.
length	데이터의 크기를 명시한다.
reference	포인터 타입으로 사용할 것인 지 지정한다.

– type_name

구성요소	설명
identifier	외부 데이터 타입을 지정하게 될 데이터 타입을 명시한다. 반드시 TYPE 문장 이전에 사용자가 지정한 데이터 타입 이름을 사용해야 한다.

– ext_datatype

구성요소	설명
STRING	STRING 타입을 지정한다.
VARRAW	VARRAW 타입을 지정한다.
LONG VARRAW	LONG VARRAW 타입을 지정한다.
VARCHAR	VARCHAR 타입을 지정한다.
VARCHAR2	VARCHAR2 타입을 지정한다.
RAW	RAW 타입을 지정한다.
LONG VARCHAR	LONG VARCHAR 타입을 지정한다.
LONG RAW	LONG RAW 타입을 지정한다.
CHARZ	CHARZ 타입을 지정한다.

● 예제

다음은 타입을 사용하는 예이다.

```
struct example_s {
    short len;
    char data[2000];
};
typedef struct example_s example_t;
EXEC SQL TYPE example_t IS VARRAW(2000);
```

8.4.40. UPDATE

UPDATE는 테이블 또는 뷰의 컬럼 값을 갱신할 때 사용하는 문장이다. 일부 컬럼 또는 전체 컬럼에 대해 갱신을 수행할 수 있다. 문장을 사용할 때 입력 변수로 배열 변수를 사용할 수도 있으며, 지시자 변수를 함께 사용할 수 있다.

UPDATE의 세부 내용은 다음과 같다.

- 문법

UPDATE의 문법에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

- 특권

UPDATE를 사용해 갱신을 수행하려면, 대상 테이블 또는 뷰에 대한 UPDATE 객체 특권을 갖거나 UPDATE ANY TABLE 시스템 특권을 가지고 있어야 한다.

- 구성요소

구성요소	설명
SET	SET 절에 입력 배열 변수가 포함되어 있다면, WHERE 절에도 반드시 같은 크기의 입력 배열 변수가 포함되어야 한다.
WHERE	WHERE 절에 입력 배열 변수가 포함되어 있다면, SET 절에도 반드시 같은 크기의 입력 배열 변수가 포함되어야 한다. UPDATE 문장을 커서와 함께 사용할 수도 있다. 현재 커서가 가리키는 로우의 컬럼 값을 갱신하려면 WHERE 절에 CURRENT OF와 커서 이름을 포함시킨다. 커서를 사용하려면 커서는 이미 열려있는 상태이어야 한다.
FOR	입력 배열 변수를 사용할 때 FOR 절을 이용하여 삽입할 데이터의 개수를 지정할 수 있다. FOR 절이 포함되어 있지 않거나 FOR 절에서 지정한 개수가 입력 배열 변수의 크기보다 크다면, 전체 배열 변수에 저장된 데이터에 대하여 갱신을 수행한다.

- 예제

다음은 UPDATE를 사용하는 예이다.

```
EXEC SQL UPDATE EMP SET SALARY = SALARY * 1.05;

EXEC SQL UPDATE EMP SET SALARY = SALARY * :ratio
    WHERE DEPTNO = :deptno;

EXEC SQL FOR :count UPDATE EMP
    SET SALARY = SALARY * :ratio_arr
```

```
WHERE DEPTNO = :deptno_arr;
```

```
EXEC SQL UPDATE EMP SET SALARY = SALARY * 1.05  
WHERE CURRENT OF emp_cursor;
```

위의 예에서 알 수 있듯이 UPDATE 문장은 EXEC SQL로 시작한다는 것을 제외하면, 일반적인 SQL 문장의 문법과 크게 다르지 않다.

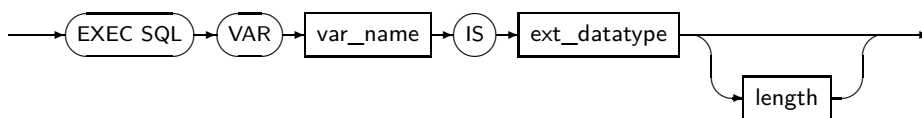
8.4.41. VAR

VAR는 특정 호스트 변수의 데이터 타입을 사용자가 지정한 데이터 타입으로 지정할 때 사용하는 문장이다. 지정하고자 하는 데이터가 호스트 변수로 반드시 선언되어 있어야 한다.

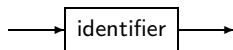
VAR의 세부 내용은 다음과 같다.

- 문법

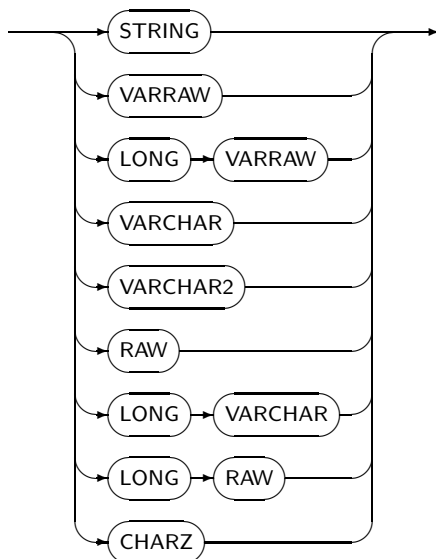
var_statement



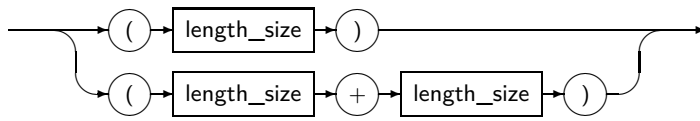
var_name



ext_datatype



length



- 구성요소

- var_statement

구성요소	설명
var_name	데이터 타입을 지정하고자 하는 호스트 변수를 명시한다.
ext_datatype	사용자가 지정할 데이터 타입을 명시한다.
length	데이터의 크기를 명시한다.

- var_name

구성요소	설명
identifier	데이터 타입을 지정하게 될 호스트 변수를 명시한다. 반드시 VAR 문장 이전에 호스트 변수로 지정된 변수명을 사용해야 한다.

- ext_datatype

구성요소	설명
STRING	STRING 타입을 지정한다.
VARRAW	VARRAW 타입을 지정한다.
LONG VARRAW	LONG VARRAW 타입을 지정한다.
VARCHAR	VARCHAR 타입을 지정한다.
VARCHAR2	VARCHAR2 타입을 지정한다.
RAW	RAW 타입을 지정한다.
LONG VARCHAR	LONG VARCHAR 타입을 지정한다.
LONG RAW	LONG RAW 타입을 지정한다.
CHARZ	CHARZ 타입을 지정한다.

- length

구성요소	설명
length_size	데이터 타입의 크기를 지정한다.

- 예제

다음은 VAR를 사용하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION;
char buffer[100];
EXEC SQL VAR buffer IS RAW(100);
EXEC SQL END DECLARE SECTION;
```

8.4.42. WHENEVER

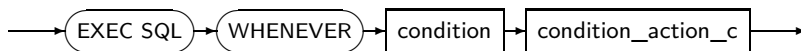
WHENEVER는 **tbESQL/C** 프로그램을 실행하는 도중, 예외 상황이 발생했을 경우에 대비하여 수행할 작업을 선언할 때 사용하는 문장이다. 발생 가능한 예외 상황에는 액세스할 로우가 없는 경우와 에러 또는 경고가 발생한 경우가 있다.

WHENEVER 문장의 효과 범위는 **WHENEVER** 문장이 명시된 위치부터 다음 **WHENEVER** 문장이 나타날 때까지이다. 하지만 문장이 한 번만 명시되었다면 효과 범위는 **WHENEVER** 문장이 명시된 위치부터 프로그램의 마지막까지이다.

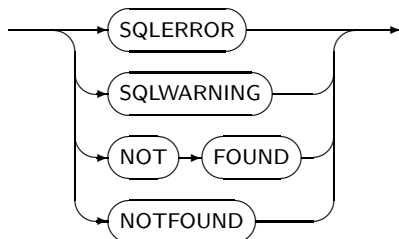
WHENEVER의 세부 내용은 다음과 같다.

- 문법

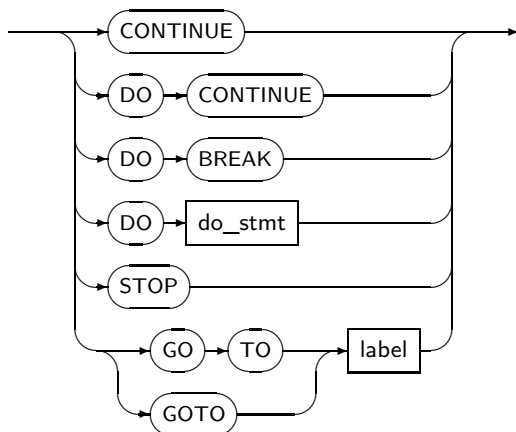
whenever_statement_c



condition



condition_action_c



- 구성요소

– whenever_statement_c

구성요소	설명
condition	에러나 경고 또는 결과 로우가 없는 등의 예외 상황을 명시한다.
condition_action_c	예외 상황이 발생했을 경우 그에 따른 대처 방법을 명시한다.

– condition

구성요소	설명
SQLERROR	에러가 발생한 경우이다.
SQLWARNING	경고가 발생한 경우이다.
NOT FOUND 또는 NOTFOUND	질의 결과 로우가 없거나 커서를 사용해 읽을 로우가 더 이상 없는 경우이다.

– condition_action_c

구성요소	설명
CONTINUE	다음 라인부터 프로그램을 계속 진행한다.
DO CONTINUE	루프 내에서 CONTINUE를 실행한다.
DO BREAK	루프 내에서 BREAK를 실행한다.
DO do_stmt	C 프로그래밍 언어로 작성한 에러 처리 문장을 수행한다.
STOP	현재 트랜잭션을 롤백하고 프로그램을 정지한다.
GOTO label	해당 label이 있는 곳으로 이동하여 프로그램을 진행한다.

- 예제

다음은 WHENEVER를 사용하는 예이다.

```
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL WHENEVER SQLERROR DO sql_error();

EXEC SQL WHENEVER SQLWARNING GOTO handle_warning;
```


제9장 tbESQL/C 프리컴파일러 옵션

본 장에서는 tbESQL/C 프리컴파일러를 동작시킬 때 사용할 수 있는 옵션에 대해 설명한다.

9.1. 개요

tbESQL/C 프리컴파일러 옵션은 프리컴파일러를 동작시킬 때 프리컴파일러가 추가적인 기능을 수행하도록 설정하기 위해서 사용된다.

프리컴파일러를 사용하려면, 다음과 같이 입력한다.

[예 9.1] tbESQL/C 프리컴파일 실행

```
tbpc test1.tbc
```

위 예를 보면 **tbpc** 유틸리티를 통해 프리컴파일을 하게 된다. '**test1.tbc**'은 프리컴파일을 실행할 대상 파일의 이름이다. 프리컴파일을 실행할 때 대상 파일의 확장자가 '.tbc'일 경우 확장자를 생략하고 사용해도 무방하다. '**test1.tbc**'는 tbESQL/C에서 제공하는 프리컴파일러의 옵션 중 하나인 INAME에 해당한다. 모든 프리컴파일러의 옵션 중에서 옵션의 이름을 생략할 수 있는 것은 INAME 하나뿐이다.

따라서 INAME을 생략하지 않고, 다음과 같이 사용해도 위의 예와 동일한 의미를 갖는다.

```
tbpc INAME=test1.tbc
```

다음은 INAME과 INCLUDE 옵션을 사용한 예이다.

[예 9.2] tbESQL/C 프리컴파일러의 옵션을 사용한 프리컴파일 실행

```
tbpc test1.tbc INCLUDE=/home/tibero/include
```

위의 예는 tbESQL/C에서 제공하는 프리컴파일러 옵션 2개를 함께 사용한 것이다. 앞서 언급했듯이 'INAME='은 생략할 수 있다. INCLUDE 옵션은 'test1.tbc' 안에서 사용된 각종 Include 파일이 같은 디렉터리에 있지 않을 경우 프리컴파일을 실행할 때 에러를 발생하게 된다. 따라서 이런 Include 파일들이 존재하는 디렉터리를 지정해줄 때 사용하는 옵션이다.

9.2. tbESQL/C 프리컴파일러 옵션 지정

프리컴파일러 옵션은 명령 프롬프트에서 직접 입력할 수도 있지만, 환경설정 파일이나 프로그램 내부에서도 지정할 수 있다. 옵션의 종류에 따라 지정할 수 있는 장소가 다르다. 어떤 곳에서도 옵션을 지정하지 않았을 경우 기본값이 적용된다.

옵션이 적용되는 순서는 다음과 같다.

1. 기본값
2. 환경설정 파일
3. 명령 프롬프트
4. 프로그램 내부

하나의 항목만을 허용하는 동일한 한 가지 옵션에 여러 번에 걸쳐 다른 항목이 지정되었을 경우 항상 마지막에 지정된 내용만 유효하다. 또한 옵션을 프로그램 내부에서 지정할 경우 옵션의 영향 범위는 C 프로그래밍 언어의 문법에서의 변수의 영향 범위와는 무관하다. 무조건 프로그램의 소스 코드의 진행 순서에서 가장 마지막에 지정된 옵션이 적용된다. INAME의 경우는 다른 옵션과 달리, 두 번 이상 INAME이 나타날 경우 에러가 발생한다.

9.2.1. 환경설정 파일

옵션을 지정할 때 환경설정 파일을 사용하는 방법에는 다음의 두 가지가 있다.

기본 환경설정 파일

tbESQL/C에서는 기본적으로 tbpc.cfg라는 환경설정 파일을 \$TB_HOME/client/config 디렉터리에 두고 있다. 이 환경설정 파일을 수정해서 원하는 옵션을 지정할 수 있다. 만약 별도의 환경설정 파일이 지정되지 않으면 tbpc 유틸리티는 자동으로 이 환경설정 파일을 먼저 읽는다.

다음은 tbpc.cfg 파일을 사용하여 프리컴파일러 옵션을 지정하는 예이다.

```
#INCLUDE=$TB_HOME/demo/chb/new_src/OV
INCLUDE=$TB_HOME/client/include
DYNAMIC=ANSI
```

위의 예에서처럼 #을 이용해서 행 전체에 대해 주석 처리를 할 수 있다. 환경설정 파일에서도 하나의 항목만 허용하는 옵션이 두 번 이상 나타났을 경우 마지막에 지정된 옵션이 적용된다.

사용자 환경설정 파일

사용자가 환경설정 파일을 임의로 생성하여 사용할 수도 있다. 파일의 위치와 파일의 이름 등을 사용자가 임의로 정할 수 있다. 이 경우 명령 프롬프트를 통해 사용할 환경설정 파일을 지정한다. 이렇게 환경설정 파일을 지정하면, tbpc.cfg 파일은 사용되지 않는다.

다음은 사용자 환경설정 파일을 지정하는 예이다.

```
tbpc test1.tbc CONFIG=config1.cfg
```

9.2.2. 명령 프롬프트

tbESQL/C 프리컴파일러 옵션은 명령 프롬프트에서 지정할 수도 있다.

명령 프롬프트에서는 다음과 같은 형태로 옵션을 지정할 수 있다.

```
[OPTION_NAME=value]
```

다음은 명령 프롬프트에서 프리컴파일러 옵션을 지정하는 예이다.

```
tbpc test1.tbc CHAR_MAP=STRING SELECT_ERROR=NO
```

9.2.3. 프로그램 내부

프로그램 내부에서도 프리컴파일러 옵션을 지정할 수 있다.

이 방법은 프리컴파일을 실행하는 도중에 옵션을 변경하고자 할 때 유용하게 사용할 수 있다. 또한 운영체제에 따라 입력할 수 있는 글자 길이의 제한으로 인해 명령 프롬프트에서 옵션을 지정할 수 없는 경우가 있다. 이러한 경우 환경설정 파일을 사용할 수도 있지만 프로그램 내부에서 옵션을 지정할 수도 있다.

프로그램 내부에서는 다음과 같은 형태로 옵션을 지정할 수 있다.

```
EXEC TIBERO OPTION (OPTION_NAME=value)
```

프로그램 내부에서 프리컴파일러의 옵션을 지정할 때 한 가지 주의할 점은 프로그램 내부에서 지정한 옵션은 C 프로그래밍 언어의 문법에 따른 변수의 영향 범위와는 전혀 무관하다는 것이다. 즉, EXEC TIBERO OPTION으로 프로그램 내부에 준 옵션 값은 소스 프로그램에서 그 문장 이후에 나오는 tbESQL/C 문장에만 영향을 미친다.

예를 들면 다음과 같다.

```
...
EXEC TIBERO OPTION (HOLD_CURSOR=NO);
... /* 이 부분에서는 어떠한 C 프로그래밍 언어의 문법이 있더라도 그 영향 범위와는 무관하게
      모든 tbESQL/C의 HOLD_CURSOR 옵션값은 NO이다. */

EXEC TIBERO OPTION (HOLD_CURSOR=YES);
... /* 이 부분에서는 어떠한 C 프로그래밍 언어의 문법이 있더라도 그 영향 범위와는 무관하게
      모든 tbESQL/C의 HOLD_CURSOR 옵션값은 YES이다. */
```

9.3. tbESQL/C 프리컴파일러 옵션 목록

본 절에서는 Tibero에서 제공하는 프리컴파일러 옵션을 알파벳 순으로 설명한다.

다음은 tbESQL 프리컴파일러 옵션을 요약한 목록이다.

옵션	설명
CHAR_MAP	문자 배열(Character Array)을 호스트 변수로 사용할 때 남는 공간을 어떻게 채울지를 지정한다.
CLOSE_ON_COMMIT	커밋할 때 커서를 닫을 것인가를 지정한다.
CODE	전처리기로 전처리하여 생성되는 결과 파일의 코드를 지정한다.
CONFIG	옵션이 기록될 환경설정 파일을 지정한다.
CPOOL	Connection Pool 기능 사용여부를 지정한다.
CMAX	커넥션 풀에 담을 수 있는 연결의 최대 개수를 지정한다.
CMIN	커넥션 풀에서 유지할 최소 연결 개수를 지정한다.
CINCR	추가 연결이 필요할 때 한 번에 늘어나는 연결 수를 지정한다.
CTIMEOUT	풀 내부에 사용하지 않는 유휴 상태의 연결을 종료하기까지의 시간(초)을 지정한다.
CNOWAIT	CNOWAIT는 커넥션 풀의 최대 연결 수(CMAX)에 도달했을 때, 새 커넥션 요청이 대기할지(0) 아니면 즉시 실패할지(값 설정) 결정하는 옵션이다.
CPP_SUFFIX	생성되는 C++ 결과 파일 이름의 Suffix를 지정한다.
DEF_SQLCODE	SQLCODE 매크로를 생성할 것인가를 지정한다.
DEFINE	directives 전처리를 할 때 필요한 이름들을 지정한다.
DYNAMIC	Dynamic SQL 문장의 타입을 지정한다.
END_OF_FETCH	END-OF-FETCH 상황에 대한 SQLCODE 값을 지정한다.
ERROR_CODE	ESQL 애플리케이션에서 많이 사용되는 3개의 에러 코드를 Tibero와 Oracle 중에 어느 것으로 사용할지 지정한다.
HOLD_CURSOR	커서가 닫힌 후 커서 정보를 유지할 것인가를 지정한다.
IGNORE_ERROR	존재하지 않는 객체에 대한 TBR-15044 에러를 무시하고 프리컴파일을 계속 진행할지 여부를 지정한다.
INAME	프리컴파일을 실행할 파일의 이름을 지정한다.
INCLUDE	#include 파일의 경로를 지정한다.
INSERT_NO_DATA_ERROR	INSERT, SELECT 시에도 NO_DATA_FOUND 기능을 사용할 것인 지를 지정한다.
LAZY_PSTMT_CHECK	컴파일 시점에서 DECLARE CURSOR가 참조하는 Statement의 존재 여부 확인을 생략할지 지정합니다.

옵션	설명
LINES	디버깅을 위한 #line 정보의 출력 여부를 지정한다.
LOG_LVL	디버깅용 로그 생성 여부를 결정한다. RUNTIME_MODE가 ODBC인 경우에만 유효하다.
MODE	프로그램이 전반적으로 Tiberο의 형식을 따를 것인가 아니면 ANSI의 기준을 따를 것인가를 지정한다.
ONAME	출력 파일의 이름을 지정한다.
ORACA	ORACA 구조체 사용 여부를 지정한다.
PARSE	전처리를 할 때 입력 파일의 내용을 어느 범위까지 파싱할 것인가를 지정한다.
PREFETCH	프로그램의 속도 향상을 위해 몇 개의 로우를 미리 가져올지 지정한다.
RELEASE_CURSOR	커서가 닫힌 후 커서 정보의 해제 여부를 지정한다.
RUNTIME_MODE	런타임에 사용할 드라이버를 지정한다.
SELECT_ERROR	주어진 수 보다 많은 로우가 결과로 나왔을 때 에러를 발생시킬 것인가의 여부를 지정한다.
SQLCHECK	SQL 문장의 내용을 어느 범위까지 검사할 것인가를 지정한다.
SQLERRD_INIT	모든 ESQl 문장 수행 전에 SQLERRD[2] 값의 초기화 여부를 지정한다.
STMT_CACHE	Dynamic SQL 문장에 대해 cache size를 지정한다.
SYS_INCLUDE	전처리를 할 때 사용될 시스템 헤더 파일의 경로를 지정한다.
THREADS	멀티 스레드를 지원할 것인가를 지정한다.
TYPE_CODE	Dynamic SQL 문장의 방법 4를 사용하는 방법을 지정한다.
UNSAFE_NULL	지시자 변수가 없어도 NULL 값을 허용할 것인가를 지정한다.
USERID	SQLCHECK가 SEMANTICS으로 지정되었을 때 서버에 접속하기 위한 사용자 정보를 지정한다.
VARCHAR_SIZE	VARCHAR 타입의 메모리 정렬(memory alignment) 적용 여부를 지정한다.
WORDSIZE	생성되는 결과 파일을 사용할 플랫폼의 WORDSIZE를 지정한다.

9.3.1. CHAR_MAP

CHAR_MAP는 문자 배열을 호스트 변수로 사용할 때 남는 공간을 어떻게 채울지를 지정하는 옵션이다. 기본값은 CHARZ이다.

CHAR_MAP의 세부 내용은 다음과 같다.

- 문법

```
CHAR_MAP={VARCHAR2 | CHARZ | STRING | CHARF}
```

다음은 문자 배열을 호스트 변수로 사용할 때 각 항목별로 Null-terminated와 Blank-padded의 여부를 나타낸 표이다. **Null-terminated**는 실제 데이터의 마지막에 NULL문자가 삽입 되는 것을 말하고, **Blank-padded**는 실제 데이터의 마지막 이후에 남는 공간을 빈칸으로 채우는 것을 말한다.

CHAR_MAP	Null-terminated 여부	Blank-padded 여부
VARCHAR2	X	O
CHARF	X	O
CHARZ	O	O
STRING	O	X

CHARF는 VARCHAR2와 동일하지만 데이터 자체가 NULL인 실제 데이터의 남는 공간을 빈칸으로 채우지 않고 그대로 둔다.

- 지정 장소

환경설정 파일, 명령 프롬프트, 프로그램 내부

9.3.2. CLOSE_ON_COMMIT

CLOSE_ON_COMMIT은 커밋을 할 때 커서를 닫을 것인가 아니면 닫지 않을 것인가를 지정하는 옵션이다.

CLOSE_ON_COMMIT의 세부 내용은 다음과 같다.

- 문법

```
CLOSE_ON_COMMIT={YES | NO}
```

항목	설명
YES	EXEC SQL COMMIT을 실행할 때 해당 트랜잭션이 처리되는 동안에 열려있던 커서를 자동으로 닫는다.
NO	EXEC SQL CLOSE cursor_name을 사용해서 사용자가 직접 열린 커서를 닫아야 한다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.3. CODE

CODE는 전처리기로 전처리하여 생성되는 결과 파일의 코드를 지정하는 옵션이다.

CODE의 세부 내용은 다음과 같다.

- 문법

```
CODE={C | CPP}
```

항목	설명
C	프리컴파일러는 C 코드를 생성한다. (기본값)
CPP	프리컴파일러는 C++ 코드를 생성한다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.4. CONFIG

CONFIG는 옵션이 기록될 환경설정 파일을 지정하는 옵션이다.

CONFIG의 세부 내용은 다음과 같다.

- 문법

```
CONFIG=filename
```

항목	설명
filename	옵션이 기록될 환경설정 파일의 이름을 명시한다. 기본값은 \$TB_HOME/client/config 디렉터리의 tbpc.cfg 파일이다.

- 지정 장소

명령 프롬프트

9.3.5. CPOOL

CPOOL은 이 옵션에 따라, 프리컴파일러 연결 풀 기능을 사용하거나 사용하지 않도록 설정하는 적절한 코드를 생성하는 옵션이다.

CPOOL의 세부 내용은 다음과 같다.

- 문법

CPPOOL={YES | NO}

항목	설명
YES	connection pooling을 지원한다.
NO	다른 connection pooling 옵션들은 프리컴파일러에 의해 무시된다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.6. CMAX

CMAX는 데이터베이스에 대해 열 수 있는 물리 연결의 최대 개수를 지정하는 옵션이다. 이 값에 도달하면 더 이상 물리 연결을 열 수 없다. **CMAX** 값은 항상 **CMIN**+**CINCR** 이상이어야 한다. 이 옵션들은 **CPPOOL=YES**로 설정된 경우에만 의미가 있다.

- 문법

CMAX=integer

항목	설명
integer	유효 범위: 1 ~ 65535 기본값: 100 데이터베이스에 대해 동시에 열 수 있는 물리 연결의 최대 개수를 지정한다. 일반적인 애플리케이션에서는 100개의 동시 작업으로 충분하며, 필요에 따라 적절히 설정한다.

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

CPPOOL=YES일 때만 적용된다.

9.3.7. CMIN

CMIN는 커넥션 풀에서 유지할 최소 물리 연결 개수를 지정한다. 초기에는 **CMIN** 개수만큼의 물리 연결이 미리 열리고, 이후에는 필요할 때만 추가로 연결을 연다. 최적의 성능을 위해 애플리케이션에서 동시에 실행

행될 계획/예상 문장 수에 맞게 설정하는 것이 좋다. 이 옵션들은 **CPOOL=YES**로 설정된 경우에만 의미가 있다.

- 문법

CMIN=integer

항목	설명
integer	유효 범위: 1 ~ (CMAX - CINCR) 기본값: 1 커넥션 풀에서 유지할 최소 물리 연결 개수.

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

CPOOL=YES일 때만 적용된다.

9.3.8. CINCR

CINCR는 현재 열린 물리 연결 수가 **CMAX**보다 작은 경우, 데이터베이스에 추가로 열 물리 연결의 증가 단위를 지정한다. 불필요한 연결 생성을 피하기 위해 기본값은 1이다. 이 옵션들은 **CPOOL=YES**로 설정된 경우에만 의미가 있다.

- 문법

CINCR=integer

항목	설명
integer	유효 범위: 1 ~ (CMAX - CMIN) 기본값: 1 추가로 열 물리 연결의 증가 단위.

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

CPOOL=YES일 때만 적용된다.

9.3.9. CTIMEOUT

CTIMEOUT는 지정된 시간(초) 이상 유휴 상태인 물리 연결을 종료하여, 열린 물리 연결의 수를 적정하게 유지하기 위한 옵션이다. 설정하지 않으면 유휴 연결을 타임아웃하지 않는다. 이 옵션들은 CPOOL=YES로 설정된 경우에만 의미가 있다.

- 문법

```
CTIMEOUT=integer
```

항목	설명
integer	유효 범위: 1 ~ 65535 기본값: 0 (미설정, 타임아웃하지 않음) 지정된 시간(초) 이상 유휴 상태인 물리 연결을 종료한다.

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

CPOOL=YES일 때만 적용된다. 새로운 물리 연결 생성에는 서버 왕복 비용이 수반된다.

9.3.10. CNOWAIT

CNOWAIT는 커넥션 풀에서 모든 물리 연결이 바쁘고 총 물리 연결 수가 이미 최대치(CMAX)에 도달한 상황에서, 즉시 오류를 반환할지 아니면 자유 연결이 생길 때까지 대기할지를 결정한다. 기본적으로 설정하지 않으면 대기한다. 이 옵션은 CPOOL=YES로 설정된 경우에만 의미가 있다.

- 문법

```
CNOWAIT=integer
```

항목	설명
integer	유효 범위: 1 ~ 65535 기본값: 0 (미설정, 대기)

항목	설명
	설정 시, 사용 가능한 연결이 없고 더 이상 생성할 수 없으면 즉시 에러를 반환한다. 미설정 시에는 자유 연결을 획득할 때까지 대기한다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.11. CPP_SUFFIX

CPP_SUFFIX 옵션은 CODE=CPP 옵션이 지정되어 있을 때 생성되는 C++ 파일 이름 앞에 붙여줄 SUFFIX를 지정하는 옵션이다.

CPP_SUFFIX의 세부 내용은 다음과 같다.

- 문법

```
CPP_SUFFIX=filename_extension
```

다음은 CPP_SUFFIX의 항목에 대한 설명이다.

항목	설명
filename_extension	예를 들어 CPP_SUFFIX=cpp 또는 CPP_SUFFIX=cc이다. (기본값: 없음)

- 지정 장소

명령 프롬프트

9.3.12. DEF_SQLCODE

DEF_SQLCODE는 프리컴파일 후 생성된 코드에 SQLCODE 매크로를 포함할 것인지를 지정하는 옵션이다.

DEF_SQLCODE의 세부 내용은 다음과 같다.

- 문법

```
DEF_SQLCODE={YES | NO}
```

항목	설명
YES	SQLCODE 매크로를 포함시킨다.
NO	SQLCODE 매크로를 포함시키지 않는다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

DEF_SQLCODE를 'YES'로 설정하면 다음의 코드가 생성된다.

```
#define SQLCODE sqlca.sqlcode
```

위의 sqlca.sqlcode는 SQL 문장의 수행 결과를 나타내는 상태 변수이다.

sqlca.sqlcode를 사용하기 위해서는 다음의 두 가지 방법 중 한 가지를 사용해 헤더 파일을 프로그램 소스 코드에 추가해야 한다. 만약 이 두 가지 방법 중 하나라도 헤더 파일에 포함시키지 않는다면, **tbESQL/C** 프로그램을 프리컴파일할 때 에러가 발생한다.

- 프로그램 내부

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

- 명령 프롬프트

```
EXEC SQL INCLUDE SQLCA
```

9.3.13. DEFINE

DEFINE 옵션은 전처리를 할 때 필요한 매크로 이름을 지정하는 옵션이다.

DEFINE의 세부 내용은 다음과 같다.

- 문법

```
DEFINE=name
```

다음은 DEFINE의 항목에 대한 설명이다.

항목	설명
name	이 명령행 옵션은 코드 내에 directives에 대해 처리해준다.

- 지정 장소

명령 프롬프트

9.3.14. DYNAMIC

DYNAMIC은 Dynamic SQL 문장의 타입을 지정하는 옵션이다.

DYNAMIC의 세부 내용은 다음과 같다.

- 문법

```
DYNAMIC={ANSI | ISO | TIBERO | ORACLE}
```

항목	설명
ANSI	ANSI 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 TIBERO 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ISO	ANSI와 같은 결과를 갖는다.
TIBERO	TIBERO 타입의 동적 SQL을 사용하도록 지정한다. (기본값) 이 옵션 값이 지정되었을 경우 ANSI 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ORACLE	TIBERO와 같은 결과를 갖는다.

ANSI 타입과 ISO 타입은 실제로 동작 방법이 완전히 일치한다. 또한 TIBERO 타입과 ORACLE 타입도 동작 방법이 같다. Dynamic SQL 문장에 대한 자세한 내용은 “제6장 Dynamic SQL”을 참고한다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.15. END_OF_FETCH

END_OF_FETCH는 SQL 문장의 수행 후 END-OF-FETCH 상황에서 사용자에게 반환할 SQLCODE 값을 지정하는 옵션이다.

END_OF_FETCH의 세부 내용은 다음과 같다.

- 문법

```
END_OF_FETCH={1403 | 100}
```

항목	설명
1403	설정하지 않을 경우 1403이 SQLCODE 값이 된다. (기본값)
100	ANSI 모드 또는 사용자를 지정할 경우 100이 SQLCODE 값이 된다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.16. ERROR_CODE

ERROR_CODE는 ESQL Application에서 많이 사용되는 3개의 에러 코드를 Tibero와 Oracle 중에 어느 것으로 사용할지 지정한다.

관련된 에러 코드는 다음과 같다.

- No data found: Tibero -9072, Oracle -1403.
- Indicator variable was not specified in INTO clause: Tibero -9094, Oracle -1405.
- Unique Constraint violation: Tibero -10007, Oracle -1.

ERROR_CODE의 사용법은 다음과 같다.

- 문법

```
ERROR_CODE={ TIBERO | ORACLE }
```

항목	설명
TIBERO	Tibero 에러 코드를 사용한다.
ORACLE	Oracle 에러 코드를 사용한다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.17. HOLD_CURSOR

HOLD_CURSOR는 커서가 닫힌 후 커서 정보를 유지할 것인가를 지정하는 옵션이다.

HOLD_CURSOR의 세부 내용은 다음과 같다.

- 문법

```
HOLD_CURSOR={YES | NO}
```

항목	설명
YES	커서가 닫힌 후 커서의 정보를 유지한다.
NO	커서가 닫힌 후 커서의 정보를 삭제한다. (기본값)

- 지정 장소

명령 프롬프트, 프로그램 내부

9.3.18. IGNORE_ERROR

IGNORE_ERROR는 **SQLCHECK**가 SEMANTICS 또는 FULL로 설정된 경우, 존재하지 않는 객체(프로시저, 함수 등)를 참조할 때 발생하는 TBR-15044 에러를 무시하고 프리컴파일을 계속 진행할지 여부를 지정하는 옵션이다.

IGNORE_ERROR의 세부 내용은 다음과 같다.

- 문법

```
IGNORE_ERROR={YES | NO}
```

항목	설명
YES	TBR-15044 에러가 발생해도 프리컴파일을 계속 진행한다. 에러 메시지는 여전히 출력되지만 프리컴파일은 성공적으로 완료된다.
NO	TBR-15044 에러가 발생하면 프리컴파일을 중단한다. (기본값)

- 지정 장소

명령 프롬프트

- 사용법

PSM 블록에서 존재하지 않는 프로시저나 함수를 호출하는 경우, TBR-15044 에러를 무시하고 오라클 Pro*C와 유사한 동작을 위해 이 옵션을 YES로 설정할 수 있다.

```
EXEC SQL EXECUTE
  BEGIN
    RefreshMaterializedView(); /* 존재하지 않는 프로시저 */
  END;
END-EXEC;
```

9.3.19. INAME

INAME은 프리컴파일을 실행할 대상 파일을 지정하는 옵션이다.

INAME의 세부 내용은 다음과 같다.

- 문법

```
INAME=filename
```

다음은 INAME의 항목에 대한 설명이다.

항목	설명
filename	프리컴파일을 실행할 대상 파일의 이름을 명시한다. (기본값: 없음)

- 지정 장소

명령 프롬프트

- 사용법

파일의 이름을 지정할 때 'INAME=' 부분'과 파일의 확장자가 tbc일 경우 .tbc도 생략할 수 있다. 따라서 다음의 예는 모두 같은 의미이다.

```
tbpc INAME=test1.tbc
tbpc test1.tbc
tbpc test1
```

9.3.20. INCLUDE

INCLUDE는 Include 파일이 위치한 경로를 지정하는 옵션이다. 프로그램 소스 내부에 사용된 각종 Include 파일이 같은 디렉터리에 있지 않은 경우 프리컴파일할 때 에러가 발생한다. 따라서 Include 파일이 존재하는 디렉터리를 지정해 주어야 하며, 그때 이 옵션을 사용한다.

INCLUDE의 세부 내용은 다음과 같다.

- 문법

```
INCLUDE=pathname
```

항목	설명
pathname	Include 파일이 존재하는 디렉터리를 명시한다. (기본값: 없음)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.21. INSERT_NO_DATA_ERROR

현재 no data found 처리는 UPDATE와 DELETE 문에 대해서만 적용되고 있다. INSERT_NO_DATA_ERROR는 insert, select 구문을 사용하는 경우에도 no data found 처리를 할 것인지 지정하는 옵션이다.

INSERT_NO_DATA_ERROR의 세부 내용은 다음과 같다.

- 문법

```
INSERT_NO_DATA_ERROR={YES | NO}
```

항목	설명
YES	EXEC SQL INSERT, SELECT 사용 시에 선택된 row가 없으면 NO_DATA_FOUND를 반환한다.
NO	선택된 row가 없어도 NO_DATA_FOUND 처리 없이 진행된다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.22. LAZY_PSTMT_CHECK

LAZY_PSTMT_CHECK는 컴파일 시점에서 DECLARE CURSOR 구문이 참조하는 Statement의 존재 여부를 확인하지 않는 옵션이다. 이 옵션을 YES로 설정하면 DECLARE CURSOR와 PREPARE 구문의 실행 순서에 관계없이 코드를 작성할 수 있다. 그러나 컴파일 단계에서 PREPARE 구문의 누락을 감지할 수 없어 런타임에 정의되지 않은 동작(undefined behavior)이 발생할 위험이 있다. 따라서 OPEN 구문을 실행하기 전에 반드시 PREPARE 구문을 포함해야 한다. 참고로 EXECUTE 구문에는 이 옵션이 적용되지 않는다.

LAZY_PSTMT_CHECK의 세부 내용은 다음과 같다.

- 문법

```
LAZY_PSTMT_CHECK={YES | NO}
```

항목	설명
YES	컴파일 시점에서 DECLARE CURSOR 구문이 참조하는 Statement 존재 여부를 확인하지 않는다. DECLARE, PREPARE 순서로 사용 가능하지만, OPEN 이전에 PREPARE를 하지 않으면 컴파일은 문제없지만 런타임 시점에 Undefined Behavior가 발생할 수 있다.
NO	컴파일 시점에서 Declare CURSOR 구문이 참조하는 Statement 존재 여부를 확인한다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

```
# tbpc LAZY_PSTMT_CHECK=YES dynamic_method3.tbc

int main() {
    EXEC SQL BEGIN DECLARE SECTION;
        ...
        char *sql_string = "SELECT ename FROM emp WHERE empno = :id";
        ...
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL DECLARE emp_cursor CURSOR FOR stmt;
    EXEC SQL PREPARE stmt FROM :sql_string; // LAZY_PSTMT_CHECK=NO 이면 PREPARE
                                           // 문장이 DELCARE CURSAOR 문장보다 앞에
                                           // 있어야 한다.

    ...

    EXEC SQL OPEN emp_cursor USING :emp_ids; // PREPARE 문장이 꼭 OPEN 문장보다
                                           // 앞에 있어야 한다.

    ...

    return 0;
}
```

9.3.23. LINES

LINES는 디버깅을 위한 #line 정보를 프리컴파일의 결과 파일에 출력할 것인지 여부를 지정하는 옵션이다.

LINES의 세부 내용은 다음과 같다.

- 문법

```
LINES={YES | NO}
```

항목	설명
YES	#line 정보를 출력한다.

항목	설명
	YES로 설정하고 프리컴파일할 경우 생성되는 파일(일반적으로 .c 파일)에 #line 정보를 추가된다. 따라서 GDB 등의 디버깅 툴을 사용할 때 원본 파일(일반적으로 .tbc 파일)을 참고하면서 디버깅 작업을 할 수 있다.
NO	#line 정보를 출력하지 않는다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.24. LOG_LVL

LOG_LVL은 디버깅용 로그 생성 여부를 결정하는 옵션으로, **RUNTIME_MODE**가 ODBC인 경우에만 유효하다. 구체적으로 ESQL 런타임에서 tbertl_odbc 라이브러리를 사용할 때만 동작한다.

LOG_LVL의 세부 내용은 다음과 같다.

- 문법

```
LOG_LVL={NONE | TRACE | DEBUG}
```

항목	설명
NONE	로그를 생성하지 않는다. (기본값)
TRACE	TRACE 레벨의 로그를 stdout으로 출력한다.
DEBUG	DEBUG 레벨의 로그를 stdout으로 출력한다.

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

이 옵션은 **RUNTIME_MODE**=ODBC로 설정된 경우에만 적용된다. stdout에 로그가 출력된다.

ODBC아닌 경우에 로그를 출력하는 방법은 **ESQL_LOG_DIR**과 **ESQL_LOG_LVL** 환경변수를 사용하여 로그 파일을 생성이 가능하며, 이 경우에 stdout에 로그가 출력되는 것이 아니라 파일에 저장된다.

다음은 LOG_LVL=TRACE로 설정했을 때의 사용 예이다.

```
tbspc RUNTIME_MODE=ODBC LOG_LVL=TRACE sample.tbc
gcc -o sample sample.c -I$TB_HOME/client/include -L$TB_HOME/client/lib \
-ltbertl_odbc -lpthread -lm -ltbodbc
./sample
```

위와 같이 설정하면 다음과 같은 형태의 로그가 stdout으로 출력된다.

```
esql_log[TRACE] ertl_odbc initialized
esql_log[DEBUG] ESQL_EXECUTE:stmt[DROP TABLE IF EXISTS emp] stmt_type[12]
esql_log[TRACE] SQLExecDirect:hstmt[50331654] stmt[DROP TABLE IF EXISTS emp]
esql_log[DEBUG] EXECUTE ERROR_OCCURED:err_code[-509073]
```

9.3.25. MODE

MODE는 tbESQL/C 프로그램이 전반적으로 Tibero의 형식을 따를 것인지 아니면, ANSI의 기준을 따를 것인지를 지정하는 옵션이다. 이 옵션은 여러 가지 옵션을 한꺼번에 지정해 주는 역할을 한다. 이 옵션으로 CLOSE_ON_COMMIT, DYNAMIC, TYPE_CODE 등의 옵션 값을 한꺼번에 지정할 수 있으며, 추가로 다른 조건을 지정할 수도 있다.

MODE의 세부 내용은 다음과 같다.

- 문법

```
MODE={ANSI | ISO | TIBERO | ORACLE}
```

항목	설명
ANSI	ANSI 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 Tibero 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ISO	ANSI와 같은 결과를 갖는다.
TIBERO	Tibero 타입의 동적 SQL을 사용하도록 지정한다. (기본값) 이 옵션 값이 지정되었을 경우 ANSI 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ORACLE	TIBERO와 같은 결과를 갖는다.

다음은 MODE 옵션에 지정된 값에 따라 설정되는 세부 옵션이다.

옵션	ANSI	TIBERO
CLOSE_ON_COMMIT	YES	NO
DYNAMIC	ANSI	TIBERO
TYPE_CODE	ANSI	TIBERO

ANSI는 ISO와, TIBERO는 ORACLE과 동일한 기능을 한다.

- 지정 장소

9.3.26. ONAME

ONAME은 프리컴파일러의 결과물로 나오는 출력 파일의 이름을 지정하는 옵션이다.

ONAME의 세부 내용은 다음과 같다.

- 문법

```
ONAME=filename
```

항목	설명
filename	원하는 출력 파일의 이름을 명시한다. (기본값: 입력된 파일 이름에서 확장자만 .c로 바꾼다.)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.27. ORACA

ORACA는 Oracle Communications Area 구조체를 사용할지 여부를 지정하는 옵션이다.

ORACA의 세부 내용은 다음과 같다.

- 문법

```
ORACA={YES | NO}
```

항목	설명
YES	프로그램 내에 EXEC SQL INCLUDE ORACA나 #include oraca.h 문장이 존재해야 한다.
NO	기본값이다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.28. PARSE

PARSE는 `tbESQL/C` 문장의 내용을 전처리할 때 어느 범위까지 파싱할 것인가를 지정하는 옵션이다.

PARSE의 세부 내용은 다음과 같다.

- 문법

```
PARSE={NONE | PARTIAL | FULL}
```

항목	설명
NONE	EXEC SQL {BEGIN END} DECLARE SECTION 안에 있는 호스트 변수 선언들과 매크로 명령들만을 처리한다.
PARTIAL	EXEC SQL {BEGIN END} DECLARE SECTION 안에 있는 호스트 변수 선언들과 모든 매크로 명령들을 처리한다. Windows 환경과 C++ 언어를 사용할 때는 반드시 이 값을 사용해야 한다.
FULL	전처리기에 내장되어 있는 <code>c parser</code> 가 EXEC SQL {BEGIN END} DECLARE SECTION 안에 뿐만 아니라 밖에 선언된 변수들도 처리를 하며, <code>include</code> 되는 헤더 파일의 매크로, 변수까지도 처리를 한다. (기본값)

- 지정 장소

명령 프롬프트, 프로그램 내부

9.3.29. PREFETCH

PREFETCH는 속도 향상을 위해 몇 개의 로우를 미리 가져올지를 지정하는 옵션이다.

PREFETCH의 세부 내용은 다음과 같다.

- 문법

```
PREFETCH=integer
```

항목	설명
integer	미리 가져올 로우의 개수를 지정한다. (기본값: 1)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.30. RELEASE_CURSOR

RELEASE_CURSOR는 커서가 닫힌 후 커서에 저장된 정보의 해제 여부를 지정하는 옵션이다.

RELEASE_CURSOR의 세부 내용은 다음과 같다.

- 문법

```
RELEASE_CURSOR={YES | NO}
```

항목	설명
YES	커서가 닫히면 커서의 정보를 해제한다.
NO	커서가 닫혀도 커서의 정보를 해제하지 않는다. (기본값)

- 지정 장소

명령 프롬프트, 프로그램 내부

9.3.31. RUNTIME_MODE

RUNTIME_MODE는 런타임에 사용할 드라이버를 지정하는 옵션이다.

RUNTIME_MODE의 세부 내용은 다음과 같다.

- 문법

```
RUNTIME_MODE={ODBC | TIBERO}
```

항목	설명
ODBC	odbc 함수를 사용한다. 컴파일과 링크 과정에서 tbERTL 라이브러리 이외에 odbc 라이브러리를 함께 링크해야 한다.
TIBERO	tbCLI 함수를 사용한다. 컴파일과 링크 과정에서 tbERTL 라이브러리 이외에 tbCLI 라이브러리를 함께 링크해야 한다. (기본값)

- 지정 장소

명령 프롬프트, 프로그램 내부

9.3.32. SELECT_ERROR

SELECT_ERROR는 호스트 변수 등으로 인해 주어진 수행 결과 개수보다 실제로 질의를 수행한 결과가 더 많이 반환된 경우 에러를 발생시킬 것인가를 지정하는 옵션이다.

SELECT_ERROR의 세부 내용은 다음과 같다.

- 문법

```
SELECT_ERROR={YES | NO}
```

항목	설명
YES	더 많은 로우가 반환되었을 경우 에러를 발생시킨다. (기본값)
NO	더 많은 로우가 반환되어도 에러를 발생시키지 않는다.

- 지정 장소

명령 프롬프트, 프로그램 내부

- 사용법

다음은 tbESQL/C 프로그램의 예이다.

```
char EMPNO[4];
char PNO[4];

EXEC SQL SELECT EMPNUM, PNUM
        INTO   :EMPNO, :PNO
        FROM   WORKS
        WHERE  EMPNUM = 'E3';
```

위 예에서 만약 질의를 수행한 결과가 더 많은 로우를 반환하는 경우 SELECT_ERROR 옵션에 지정된 값에 따라 다음과 같이 나타나는 결과가 다르다.

항목	결과
YES	ERROR_ESQL_TOO_MANY_ROW_IN_SELECT 에러가 발생한다.
NO	수행된 질의 결과의 첫 번째 로우를 호스트 변수에 할당한다. 이때 에러는 발생하지 않는다.

9.3.33. SQLCHECK

SQLCHECK는 tbESQL/C 문장의 내용을 어느 범위까지 검사할 것인가를 지정하는 옵션이다.

SQLCHECK의 세부 내용은 다음과 같다.

- 문법

```
SQLCHECK={SEMANTICS | FULL | SYNTAX}
```

항목	설명
SEMANTICS	SEMANTICS로 지정되면, 프리컴파일러는 서버에 접속을 시도한다. 따라서 SEMANTICS로 설정하기 위해서는 USERID 옵션이 반드시 필요하다. 접속이 성공하면 프리컴파일러는 SQL 문장을 서버로 보내 문법뿐만 아니라 SQL 문장이 참조하는 스키마 객체의 내용까지 검사하며, 컬럼의 타입 정보 등을 바탕으로 올바른 타입끼리 호스트 변수에 바인딩 되는지도 검사한다. tbESQL/C 문장뿐만 아니라 DDL이나 PSM일 경우에도 문법과 내용을 모두 검사한다.
FULL	SEMANTICS와 동일한 의미를 갖는다.
SYNTAX	tbpc 유틸리티는 서버에 접속하지 않고 자체적으로 ESQL 문법과 SQL 문법을 검사한다. (기본값)

- 지정 장소

명령 프롬프트, 프로그램 내부

9.3.34. SQLERRD_INIT

SQLERRD_INIT은 모든 ESQL 문장의 수행 전에 이전 ESQL 문장 수행에 의해 남아있는 SQLCA.SQLERRD[2] 값의 초기화 여부를 지정하는 옵션이다.

SQLERRD_INIT의 세부 내용은 다음과 같다.

- 문법

```
SQLERRD_INIT={YES | NO}
```

항목	설명
YES	SQLCA.SQLERRD[2]의 값을 초기화한다. (기본값)
NO	SQLCA.SQLERRD[2]의 값을 초기화하지 않는다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.35. STMT_CACHE

STMT_CACHE는 Dynamic SQL 문장에 대해 cache size를 지정하는 옵션이다.

STMT_CACHE의 세부 내용은 다음과 같다.

- 문법

```
STMT_CACHE=integer
```

항목	설명
integer	애플리케이션에서 distinct한 Dynamic SQL 문에 대해 cache size를 지정한다. (기본값: 10)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.36. SYS_INCLUDE

SYS_INCLUDE는 system header 파일이 위치한 경로를 지정하는 옵션이다.

SYS_INCLUDE의 세부 내용은 다음과 같다.

- 문법

```
SYS_INCLUDE=pathname
```

항목	설명
pathname	system header 파일이 존재하는 디렉토리를 명시한다. (기본값: 없음)

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.37. THREADS

THREADS는 멀티 스레드를 지원 여부를 지정하는 옵션이다.

THREADS의 세부 내용은 다음과 같다.

- 문법

```
THREADS={YES | NO}
```

항목	설명
YES	멀티 스레드를 지원한다.
NO	멀티 스레드를 지원하지 않는다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

다음과 같은 문장에 멀티 스레드를 사용하려면, THREADS 옵션의 값은 반드시 'YES'로 설정되어 있어야 한다.

– [ENABLE THREADS](#)

– [CONTEXT USE](#)

– [CONTEXT ALLOCATE](#)

– [CONTEXT FREE](#)

9.3.38. TYPE_CODE

TYPE_CODE는 Dynamic SQL 문장을 사용하는 방법 중에 방법 4를 사용하는 방법을 지정하는 옵션이다. Dynamic SQL 문장에 대한 자세한 내용은 “[제6장 Dynamic SQL](#)”을 참고한다.

TYPE_CODE의 세부 내용은 다음과 같다.

- 문법

```
TYPE_CODE={ANSI | ISO | TIBERO | ORACLE}
```

항목	설명
ANSI	ANSI 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 Tiberio 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ISO	ANSI와 같은 결과를 갖는다.
TIBERO	Tiberio 타입의 동적 SQL을 사용하도록 지정한다. (기본값) 이 옵션 값이 지정되었을 경우 ANSI 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ORACLE	TIBERO와 같은 결과를 갖는다.

ANSI 타입은 ISO 타입과 동작 방법이 동일하며, TIBERO 타입은 ORACLE 타입과 동작 방법이 동일하다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.39. UNSAFE_NULL

UNSAFE_NULL은 지시자 변수가 없을 때 NULL 값을 허용할 것인지의 여부를 지정하는 옵션이다. NULL 데이터가 예상되는 경우에도 편의상 지시자 변수 사용을 생략하고 싶을 때 이 옵션을 사용한다.

UNSAFE_NULL의 세부 내용은 다음과 같다.

- 문법

```
UNSAFE_NULL={YES | NO}
```

항목	설명
YES	NULL을 허용한다. 지시자 변수 없이 NULL 데이터를 호스트 변수로 받아도 에러는 발생하지 않는다.
NO	NULL을 허용하지 않는다. (기본값) 지시자 변수 없이 NULL 데이터를 호스트 변수로 받으면 1405 에러를 발생한다. 1405 에러는 가져온 데이터가 NULL이나 지시자 변수가 존재하지 않아 tbESQL/C 런타임 라이브러리가 개발자에게 NULL 값의 존재 여부를 알려 줄 방법이 없을 경우 발생하는 에러이다.

- 지정 장소

환경설정 파일, 명령 프롬프트

9.3.40. USERID

USERID는 서버 접속을 할 때 필요한 사용자 계정의 정보를 지정하는 옵션이다. 이 옵션은 SQL 문장의 내용을 검사하기 위한 접속 정보일 뿐이며, 실제 데이터베이스를 실행할 때의 접속 정보와는 무관하다.

USERID의 세부 내용은 다음과 같다.

- 문법

```
USERID=username/password
```

항목	설명
username	사용자의 이름을 명시한다.
password	사용자의 패스워드를 명시한다.

- 지정 장소

명령 프롬프트

- 사용법

SQLCHECK를 SEMANTICS나 FULL로 지정할 경우 반드시 이 옵션을 사용해야 한다.

9.3.41. VARCHAR_SIZE

VARCHAR_SIZE는 VARCHAR 타입의 메모리 정렬(memory alignment)을 적용할지 여부를 지정하는 옵션이다. 이 옵션을 YES로 설정하면 효율적인 메모리 사용을 위해 VARCHAR 생성 크기가 조정된다.

VARCHAR_SIZE의 세부 내용은 다음과 같다.

- 문법

```
VARCHAR_SIZE={YES | NO}
```

항목	설명
YES	VARCHAR 크기에 대해 메모리 정렬을 적용한다. 선언된 크기를 4바이트 경계에 맞게 조정하여 효율적인 메모리 사용을 제공한다.
NO	VARCHAR 크기에 대해 메모리 정렬을 적용하지 않는다. 선언된 크기 그대로 사용한다. (기본값)

- 지정 장소

환경설정 파일, 명령 프롬프트

- 사용법

VARCHAR_SIZE=YES로 설정하면 다음과 같은 크기 조정이 적용된다.

```
/* 예제: VARCHAR_SIZE=YES인 경우 컴파일 전과 후 */
varchar a[5];
=> struct __tag07 { unsigned short len; unsigned char arr[6]; } a;

/* 수식도 계산됨 */
```

```
varchar b[14+1];
=> struct __tag08 { unsigned short len; unsigned char arr[18]; } b;
```

크기 조정 규칙: (조정된 크기 + 2)가 4의 배수가 되도록 조정된다.

VARCHAR_SIZE=NO인 경우 선언한 크기가 그대로 유지된다:

```
/* 예제: VARCHAR_SIZE=NO인 경우 경우 컴파일 전과 후 */
varchar a[5];
=> struct __tag07 { unsigned short len; unsigned char arr[5]; } a;
```

- 주의사항

이 옵션을 사용할 때는 다음 사항에 주의해야 한다:

- 기존 코드와의 호환성: VARCHAR_SIZE=YES로 설정하면, 기존 코드에서 개발자가 선언된 크기를 그대로 최대 크기로 가정하고 작성한 부분에 영향을 줄 수 있다.

예를 들어,

```
varchar var[5];
EXEC SQL SELECT ... INTO :var FROM ...;

printf("%.s\n", var.len, var.arr);
```

개발자는 `varchar var[5]`라고 선언했으므로 문자열 최대 길이가 5일 것이라고 생각할 수 있다. 하지만 VARCHAR_SIZE=YES일 경우 내부적으로 배열 크기가 6으로 확장된다. 따라서 SELECT 결과가 최대 6바이트까지 들어올 수 있으며, 이로 인해 출력 시 예상보다 더 큰 문자열이 처리될 수 있다.

즉, 기존에 "최대 길이 = 선언 크기"라고 가정한 로직은 깨질 수 있으므로 주의해야 한다.

9.3.42. WORDSIZE

WORDSIZE는 64bits에서 32bits용 소스 또는 그 반대로도 프리컴파일러로 소스를 생성할 수 있도록 하는 옵션이다. (기본값: 64)

WORDSIZE의 세부 내용은 다음과 같다.

- 문법

```
WORDSIZE={64 | 32}
```

- 지정 장소

환경설정 파일, 명령 프롬프트

Appendix A. tbESQL/C 프로그램 예제

본 장에서는 DDL과 PSM, 데이터베이스의 이름, 대용량 객체형을 사용하여 tbESQL/C 프로그램의 예제를 설명한다.

A.1. DDL과 PSM

다음은 DDL과 PSM을 사용하여 tbESQL/C 프로그램을 작성한 예이다.

```
void Test_call1(void)
{
    char *conn_str = "tibero/tmax";
    int num;
    int fact;

    EXEC SQL CONNECT :conn_str;

    EXEC SQL CREATE OR REPLACE FUNCTION fact(n IN INTEGER) RETURN INTEGER IS
    BEGIN
        IF (n <= 0) then return 1;
        ELSE return n * fact(n - 1);
        END IF;
    END fact;
    END-EXEC;

    num = 5;

    EXEC SQL CALL fact(:num) INTO :fact;

    EXEC SQL COMMIT WORK RELEASE;
}

void Test_call2(void)
{
    char *conn_str = "tibero/tmax";
    char x[10];
    int y;

    EXEC SQL CONNECT :conn_str;

    EXEC SQL DROP TABLE T1;
    EXEC SQL CREATE TABLE T1(x VARCHAR(10));
```

```

EXEC SQL CREATE OR REPLACE PROCEDURE test_proc(x IN VARCHAR, y IN OUT NUMBER) IS

    BEGIN
        INSERT INTO T1 VALUES(x);
        y := 2;
    END;
END-EXEC;

memset(x, 0, sizeof(x));
strcpy(x, "TIBERO");

EXEC SQL CALL test_proc(:x, :y);

memset(x, 0, sizeof(x));

EXEC SQL SELECT x INTO :x FROM T1; /* x값은 "TIBERO"   "가 된다 */

EXEC SQL COMMIT WORK RELEASE;
}

```

A.2. 데이터베이스의 이름

다음은 데이터베이스의 이름을 사용하여 tbESQL/C 프로그램을 작성한 예이다.

```

void Test_named_database(void)
{
    VARCHAR branch[5], postal[10], region[10];

    char *user_pass = "tibero/tmax";

    EXEC SQL WHENEVER SQLERROR
        DO printf("file: %s, line: %d, error: %d\n",
            __FILE__, __LINE__, sqlca.sqlcode);

    EXEC SQL DECLARE C1 DATABASE;

    EXEC SQL CONNECT :user_pass AT C1;

    EXEC SQL at C1 DECLARE C1 CURSOR FOR
        select * from branch order by branch_cd;

    EXEC SQL OPEN C1;

    memset(branch.arr, 0, 5);
    memset(postal.arr, 0, 10);
}

```

```

memset(region.arr, ' ', 10);

EXEC SQL FETCH C1 INTO :branch, :postal, :region;

memset(branch.arr, 0, 5);
memset(postal.arr, 0, 10);
memset(region.arr, ' ', 10);

EXEC SQL FETCH C1 INTO :branch, :postal, :region;

EXEC SQL CLOSE C1;

EXEC SQL AT C1 COMMIT;
}

```

A.3. 대용량 객체형

다음은 대용량 객체형을 사용하여 tbESQL/C 프로그램을 작성한 예이다. ClobLocator, BlobLocator 의 사용을 위해서는 esql_lob.h 헤더 파일을 명시해야 한다.

```

#include <esql_lob.h>

#define MAXBUFLEN 5000

char *conn_str = "tibero/tmax";

void Test_clob_write(void)
{
    ClobLocator *clob;
    unsigned int clob_len, write_amt, offset;
    unsigned char buffer[MAXBUFLEN];
    int i;

    EXEC SQL CONNECT :conn_str;

    EXEC SQL DROP TABLE CLOB_TABLE;

    EXEC SQL WHENEVER SQLERROR
        do printf("error: %d!!!\n", sqlca.sqlcode);

    EXEC SQL WHENEVER NOTFOUND
        do printf("notfound!!!\n");

    EXEC SQL CREATE TABLE CLOB_TABLE (num_col number, clob_col clob);

    EXEC SQL INSERT INTO CLOB_TABLE VALUES (1, empty_clob());
}

```

```

EXEC SQL ALLOCATE :clob;

EXEC SQL SELECT clob_col INTO :clob FROM CLOB_TABLE WHERE num_col = 1;

EXEC SQL LOB DESCRIBE :clob GET LENGTH INTO :clob_len;

write_amt = MAXBUFLen;
offset = clob_len + 1;

for (i = 0; i < MAXBUFLen; i += 25) {
    strcpy((char*)(buffer + i), "abcdefghijklmnopqrstuvwxy");
}

EXEC SQL LOB WRITE ONE :write_amt FROM :buffer
    WITH LENGTH :write_amt INTO :clob AT :offset;

EXEC SQL FREE :clob;

EXEC SQL COMMIT RELEASE;
}

void Test_clob_read(void)
{
    ClobLocator *clob;
    unsigned int clob_len, read_amt;
    unsigned char buffer[26];

    memset(buffer, 0, sizeof(buffer));

    EXEC SQL CONNECT :conn_str;

    EXEC SQL ALLOCATE :clob;

    EXEC SQL SELECT clob_col INTO :clob FROM CLOB_TABLE WHERE num_col = 1;

    EXEC SQL LOB DESCRIBE :clob GET LENGTH INTO :clob_len;

    read_amt = 25;

    EXEC SQL LOB READ :read_amt FROM :clob
        INTO :buffer WITH LENGTH :read_amt;

    EXEC SQL FREE :clob;

    EXEC SQL COMMIT RELEASE;
}

```

```

void Test_blob_write(void)
{
    BlobLocator *blob;
    unsigned int blob_len, offset;
    unsigned char buffer[MAXBUFLen * 2];
    FILE *fp;

    memset(buffer, 0, sizeof(buffer));

    EXEC SQL CONNECT :conn_str;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    EXEC SQL DROP TABLE BLOB_TABLE;

    EXEC SQL WHENEVER SQLERROR
        do printf("error: %d!!!\n", sqlca.sqlcode);

    EXEC SQL WHENEVER NOTFOUND
        do printf("notfound!!!\n");

    EXEC SQL CREATE TABLE BLOB_TABLE (blob_col blob);

    EXEC SQL ALLOCATE :blob;

    EXEC SQL INSERT INTO BLOB_TABLE VALUES (empty_blob())
        RETURNING blob_col INTO :blob;

    fp = fopen((const char *)"./test_blob.jpeg", (const char *)"r");
    fseek(fp, 0L, SEEK_END);
    blob_len = (unsigned int)ftell(fp);

    fseek(fp, 0L, SEEK_SET);
    fread((void *)buffer, (size_t)blob_len, (size_t)1, fp);
    offset = 1;

    EXEC SQL LOB WRITE ONE :blob_len FROM :buffer
        WITH LENGTH :blob_len INTO :blob AT :offset;

    fclose(fp);

    EXEC SQL FREE :blob;

    EXEC SQL COMMIT RELEASE;
}

```

```

void Test_blob_read(void)
{
    ClobLocator *blob;
    unsigned int blob_len, read_amt;
    unsigned char org_buffer[MAXBUFLen * 2];
    unsigned char buffer[MAXBUFLen * 2];
    FILE *fp;

    memset(buffer, 0, sizeof(org_buffer));
    memset(buffer, 0, sizeof(buffer));

    EXEC SQL CONNECT :conn_str;

    EXEC SQL ALLOCATE :blob;

    EXEC SQL SELECT blob_col INTO :blob FROM BLOB_TABLE;

    EXEC SQL LOB DESCRIBE :blob GET LENGTH INTO :blob_len;

    read_amt = MAXBUFLen * 2;

    EXEC SQL LOB READ :blob_len FROM :blob
        INTO :buffer WITH LENGTH :read_amt;

    fp = fopen((const char *)"TESTCASE/pc_success/test_blob.jpeg",
        (const char *)"r");
    fread((void *)org_buffer, (size_t)blob_len, (size_t)1, fp);

    fclose(fp);

    EXEC SQL FREE :blob;

    EXEC SQL COMMIT RELEASE;
}

```

색인

A

ALLOCATE DESCRIPTOR 문장, 97
ALLOCATE 문장, 96
ARRAY VARIABLE, 45
AT 절, 92

B

BINARY_DOUBLE, 9
BINARY_FLOAT, 8
Blank-padded, 156
BLOB, 9

C

CALL 문장, 98
CHAR, 8, 9, 11
CHAR_MAP, 155
CINCR, 159
CLOB, 9
CLOSE 절, 100
CLOSE_ON_COMMIT, 156
CMAX, 158
CMIN, 158
CNOWAIT, 160
CODE, 156
COMMIT 절, 100
CONFIG, 157
CONNECT 절, 101
CONTEXT ALLOCATE 절, 62, 103
CONTEXT FREE 절, 63, 104
CONTEXT USE 절, 63, 105
CONTINUE in WHENEVER, 87
CPOOL, 157
CPP_SUFFIX, 161
CTIMEOUT, 160
CURRENT OF 절, 38

D

DATE, 9, 10, 11
DEALLOCATE DESCRIPTOR 절, 105
DECLARE CURSOR 절, 106
DECLARE DATABASE 절, 107
DECLARE STATEMENT 절, 108
DECLARE 영역, 12, 25
DEF_SQLCODE, 161
DEFINE, 162
DELETE 절, 35, 109
DESCRIBE DESCRIPTOR 절, 111
DESCRIBE 절, 110
DESCRIPTOR 이름, 93
DO BREAK in WHENEVER, 87
DO CONTINUE in WHENEVER, 87
DO in WHENEVER, 87
DYNAMIC, 162
Dynamic SQL, 65
 방법 1, 66
 방법 2, 67
 방법 3, 69
 방법 4, 72

E

Embedded SQL, 1
ENABLE THREADS 절, 62, 112
END_OF_FETCH, 163
ERROR_CODE, 164
ESQL, 1
EXECUTE ... END-EXEC 절, 116
EXECUTE DESCRIPTOR 절, 114
EXECUTE IMMEDIATE 절, 117
EXECUTE 절, 113

F

FETCH DESCRIPTOR 절, 120
FETCH 절, 36, 117
FLOAT, 8
FOR UPDATE 절, 38
FOR 절, 56, 93

G

GET DESCRIPTOR 절, 121
GOTO in WHENEVER, 87

H

HOLD_CURSOR, 164

I

IGNORE_ERROR, 165
INAME, 165
INCLUDE, 166
INDICATOR, 20
INSERT 절, 34, 124
INSERT_NO_DATA_ERROR, 166
INTEGER, 8
INTO 절, 32

L

LAZY_PSTMT_CHECK, 167
LINES, 168
LOB CLOSE 절, 125
LOB CREATE TEMPORARY 절, 125
LOB DESCRIBE 절, 126
LOB FREE TEMPORARY 절, 128
LOB OPEN 절, 129
LOB READ 절, 130
LOB WRITE 절, 132
LOCK, 38
LOG_LVL, 169
LONG, 8
LONG RAW, 8

M

MODE, 170
Mutex, 61
Mutual Exclusion, 61

N

NCHAR, 8
NOT FOUND in WHENEVER, 87
Null-terminated, 156

NUMBER, 8, 10, 11
NVARCHAR, 8

O

ONAME, 171
OPEN 절, 36, 135
ORACA, 171

P

PARSE, 172
Precision, 2
Precompile, 5
Precompiler, 5
PREFETCH, 172
PREPARE 절, 137

R

RAW, 8, 10
RAWID, 10, 11
RELEASE_CURSOR, 173
ROLLBACK 절, 138
ROWID, 9, 14
Runtime Context, 61
RUNTIME CONTEXT, 61
RUNTIME_MODE, 173

S

SAVEPOINT 절, 139
Scale, 2
Scrollable Cursors, 41
SELECT 절, 32, 140
SELECT_ERROR, 173
SET DESCRIPTOR 절, 141
SET 절, 35
SQL 문장 연산자, 26
SQLCA, 29, 83
SQLCA 구조체, 83
SQLCHECK, 174
SQLCODE, 80, 81
SQLDA, 72
SQLERRD_INIT, 175
SQLERROR in WHENEVER, 87

SQLSTATE, 80, 81
 SQLWARNING in WHENEVER, 87
 STMT_CACHE, 175
 STOP BREAK in WHENEVER, 87
 STRUCTURAL ARRAY VARIABLE, 58
 STRUCTURAL INDICATOR, 22
 Synchronization, 61
 SYS_INCLUDE, 176

T

tbERTL 라이브러리, 28
 tbESQL /C 프리컴파일러 옵션
 LOG_LVL, 169
 tbESQL/C 데이터 타입, 9
 tbESQL/C 문장, 1, 91, 95
 ALLOCATE DESCRIPTOR 절, 97
 ALLOCATE 절, 96
 AT 절, 92
 CALL 절, 98
 CLOSE 절, 100
 COMMIT 절, 100
 CONNECT 절, 101
 CONTEXT ALLOCATE 절, 103
 CONTEXT FREE 절, 104
 CONTEXT USE 절, 105
 DEALLOCATE DESCRIPTOR 절, 105
 DECLARE CURSOR 절, 106
 DECLARE DATABASE 절, 107
 DECLARE STATEMENT 절, 108
 DELETE 절, 109
 DESCRIBE DESCRIPTOR 절, 111
 DESCRIBE 절, 110
 DESCRIPTOR 이름, 93
 ENABLE THREADS 절, 112
 EXECUTE ... END-EXEC 절, 116
 EXECUTE DESCRIPTOR 절, 114
 EXECUTE IMMEDIATE 절, 117
 EXECUTE 절, 113
 FETCH DESCRIPTOR 절, 120
 FETCH 절, 117
 FOR 절, 93
 GET DESCRIPTOR 절, 121

INSERT 절, 124
 LOB CLOSE 절, 125
 LOB CREATE TEMPORARY 절, 125
 LOB DESCRIBE 절, 126
 LOB FREE TEMPORARY 절, 128
 LOB OPEN 절, 129
 LOB READ 절, 130
 LOB WRITE 절, 132
 OPEN 절, 135
 PREPARE 절, 137
 ROLLBACK 절, 138
 SAVEPOINT 절, 139
 SELECT 절, 140
 SET DESCRIPTOR 절, 141
 TYPE 절, 143
 UPDATE 절, 145
 VAR 절, 146
 WHENEVER 문장, 148
 tbESQL/C 프리컴파일러 옵션, 151, 154
 CHAR_MAP, 155
 CINCR, 159
 CLOSE_ON_COMMIT, 156
 CMAX, 158
 CMIN, 158
 CNOWAIT, 160
 CODE, 156
 CONFIG, 157
 CPOOL, 157
 CPP_SUFFIX, 161
 CTIMEOUT, 160
 DEF_SQLCODE, 161
 DEFINE, 162
 DYNAMIC, 162
 END_OF_FETCH, 163
 ERROR_CODE, 164
 HOLD_CURSOR, 164
 IGNORE_ERROR, 165
 INAME, 165
 INCLUDE, 166
 INSERT_NO_DATA_ERROR, 166
 LAZY_PSTMT_CHECK, 167
 LINES, 168
 MODE, 170

ONAME, 171
ORACA, 171
PARSE, 172
PREFETCH, 172
RELEASE_CURSOR, 173
RUNTIME_MODE, 173
SELECT_ERROR, 173
SQLCHECK, 174
SQLERRD_INIT, 175
STMT_CACHE, 175
SYS_INCLUDE, 176
THREADS, 176
TYPE_CODE, 177
UNSAFE_NULL, 178
USERID, 178
VARCHAR_SIZE, 179
WORDSIZE, 180
tbpc, 151
THREADS, 62, 176
Tibero 데이터 타입, 7
Tibero와 tbESQL/C 데이터 타입, 9
TIME, 9, 10, 11
TIMESTAMP, 9, 10, 11
TO_CHAR 함수, 12
TO_DATE 함수, 12
TO_NUMBER 함수, 12
TYPE 절, 143
TYPE_CODE, 177

U

UNSAFE_NULL, 178
UPDATE 절, 35, 145
USERID, 178

V

VAR 절, 146
VARCHAR, 8, 9, 11, 15
VARCHAR 타입 입력 변수, 16, 17
VARCHAR 타입 출력 변수, 16, 17
VARCHAR 타입의 NULL 처리, 16
VARCHAR_SIZE, 179

W

WHENEVER, 29, 86
 CONTINUE, 87
 DO, 87
 DO BREAK, 87
 DO CONTINUE, 87
 GOTO, 87
 NOT FOUND, 87
 SQLERROR, 87
 SQLWARNING, 87
 STOP, 87
WHENEVER 문장, 148
WHERE 절, 33, 35
WORDSIZE, 180

ㄱ

구조체, 4, 17
구조체 배열 변수, 58
구조체 타입의 지시자, 22

ㄴ

날짜형, 7
내장 SQL, 1
내재형, 8

ㄷ

대용량 객체형, 7
데이터 타입 변환, 11
동기화, 61

ㄹ

런타임 에러 처리, 79
런타임 컨텍스트, 61

ㅁ

멀티 스레드(Multi-Thread) 프로그램, 61
문자형, 7
뮤텍스, 61

ㅂ

배열 변수, 4, 45, 47

人

상태 변수, 29, 79
상태 변수 선언, 80
상호 배제, 61
서브 클래스 코드, 80
숫자형, 7
스케일, 2, 8
스크롤 가능 커서, 41, 106
실행 문장, 91

ㅇ

입/출력 변수, 13
입력 배열 변수, 45
입력 변수, 3, 33

ㅈ

잠금, 38
정밀도, 2, 8
주석, 26
주의사항, 27
지시어, 91
지시자, 20

ㅊ

참조, 26
출력 배열 변수, 45
출력 변수, 3, 32

ㅋ

커서, 4, 36, 48
클래스 코드, 80

표

포인터, 18
프리컴파일, 5
프리컴파일러, 5
프리컴파일러 옵션, 62

ㅎ

호스트 변수, 155

