**Precompiler Command**

To run the Pro\*C precompiler, you issue the following command:

proc

The location of the precompiler differs from system to system. The system or database administrator usually defines logicals or aliases, or uses other system-specific means to make the Pro\*C executable accessible.

The INAME= argument specifies the source file to be precompiled. For example, the command

proc INAME=test\_proc

precompiles the file *test\_proc.pc*in the current directory, since the precompiler assumes that the filename extension is .*pc*. The INAME option does not have to be the first option on the command line, but if it is, you can omit the option specification. So, the command

proc myfile

is equivalent to

proc INAME=myfile

**Note**: The option names, and option values that do not name specific OS objects, such as filenames, are not case-sensitive. In the examples in this guide, option names are written in upper case, and option values are usually in lower case. When you enter filenames, including the name of the Pro\*C precompiler executable itself, always follow the case conventions used by your operating system. In UNIX, the executable is ``proc'', in lower case.

**Precompiler Options**

Many useful options are available at precompile time. They let you control how resources are used, how errors are reported, how input and output are formatted, and how cursors are managed.

The value of an option is a string literal, which represent text or numeric values. For example, for the option

... INAME=my\_test

the value is a string literal that specifies a filename. But for the option

...MAXOPENCURSORS=20

the value is numeric.

Some options take Boolean values, and you can represent these with the strings *yes* or *no*, *true* or *false*, or with the integer literals 1 or 0 respectively. For example, the option

... SELECT\_ERROR=yes

is equivalent to

... SELECT\_ERROR=true

or

... SELECT\_ERROR=1

all of which mean that SELECT errors should be flagged at run time.

The option value is always separated from the option name by an equals sign, with no whitespace around the equals sign.

**Default Values**

Many of the options have default values. The default value of an option is determined by:

* a value built in to the precompiler
* a value set in the Pro\*C *system configuration file*
* a value set in a Pro\*C *user configuration file*

For example, the option MAXOPENCURSORS specifies the maximum number of cached open cursors. The built-in precompiler default value for this option is 10. However, if MAXOPENCURSORS=32 is specified in the system configuration file, the default now becomes 32. The user configuration file could set it to yet another value, which then overrides the system configuration value. Then, if this option is set on the command line, the new command-line value takes precedence over the precompiler default, the system configuration file specification, and the user configuration file specification.

Finally, an inline specification takes precedence over all preceding defaults. See the section "Configuration Files" [[*]](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#cfiles) for more information about configuration files.

Some options, such as USERID, do not have a precompiler default value. The built-in default values for options that do have them are listed in [Table 7 - 1](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#ch-run-optab), and in the "Using the Precompiler Options" section starting [[*]](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch0705.htm#run-useopt).

**Determining Current Values**

You can interactively determine the current value for one or more options by using a question mark on the command line. For example, if you issue the command

proc ?

the complete set of options, along with their current values, is printed to your terminal. (On a UNIX system running the C shell, escape the `?' with a backslash.) In this case, the values are those built into the precompiler, overridden by any values in the system configuration file. But if you issue the command

proc config=my\_config\_file.h ?

and there is a file named *my\_config\_file.h* in the current directory, all options are listed. Values in the user configuration file supply missing values, and supersede values built-in to the Pro\*C precompiler, or values specified in the system configuration file.

You can also determine the current value of a single option, by simply specifying that option name, followed by =?. For example

proc maxopencursors=?

prints the current default value for the MAXOPENCURSORS option.

**Case Sensitivity**

In general, you can use either uppercase or lowercase for precompiler option names and values. However, if your operating system is case sensitive, like UNIX, you must specify filename values, including the name of the Pro\*C executable, using the correct combination of upper and lowercase letters.

**Configuration Files**

A configuration file is a text file that contains precompiler options. Each record (line) in the file contains one option, with its associated value or values. For example, a configuration file might contain the lines

FIPS=YES

MODE=ANSI

CODE=ANSI\_C

to set defaults for the FIPS, MODE, and CODE options.

There is a single system configuration file for each Oracle installation. The name of the system configuration file is *pmscfg.h*. The location of the file is system specific. On most UNIX systems, the file specification is *$ORACLE\_HOME/proc/pmscfg.h*.

Each Pro\*C user can have one or more private configuration files. The name of the configuration file must be specified using the CONFIG= precompiler option. See the "Using the Precompiler Options" section starting [[*]](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#run-useopt).

**Note**: You cannot nest configuration files. This means that CONFIG= is not a valid option inside a configuration file.

**What Occurs during Precompilation?**

During precompilation, Pro\*C generates C or C++ code that replaces the SQL statements embedded in your host program. The generated code contains data structures that indicate the datatype, length, and address of host variables, as well as other information required by the Oracle runtime library, SQLLIB. The generated code also contains the calls to SQLLIB routines that perform the embedded SQL operations.

**Note**: The precompiler does *not* generate calls to Oracle Call Interface (OCI) routines.

The precompiler can issue warnings and error messages. These messages have the prefix PCC-, and are described in the [*Oracle7 Server Messages*](https://docs.oracle.com/cd/A57673_01/DOC/server/doc/MSG73/toc.htm#org) manua*l*.

[Table 7 - 1](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#ch-run-optab) is a quick reference to the major precompiler options. It summarizes the section "Using the Precompiler Options" starting [[*]](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch0705.htm#run-useopt). The options that are accepted, but do not have any affect, are not included in this table.

**Scope of Options**

A precompilation unit is a file containing C code and one or more embedded SQL statements. The options specified for a given precompilation unit affect only that unit; they have no effect on other units. For example, if you specify HOLD\_CURSOR=YES and RELEASE\_CURSOR=YES for unit A but not for unit B, SQL statements in unit A run with these HOLD\_CURSOR and RELEASE\_CURSOR values, but SQL statements in unit B run with the default values.

|  |  |  |
| --- | --- | --- |
| **Syntax** | **Default** | **Specifics** |
| AUTO\_CONNECT=YES|NO | NO | Automatic OPS$ logon |
| CODE=ANSI\_C | KR\_C | CPP | KR\_C | kind of C code generated |
| COMP\_CHARSET= MULTI\_BYTE|SINGLE\_BYTE | MULTI\_BYTE | the character set type the C/C++ compiler supports |
| CONFIG=<filename> | none | user's private configuration file |
| CPP\_SUFFIX=<extension> | none | specify the default filename extension for output files |
| DBMS=V6 | V7 | NATIVE|V6\_CHAR | NATIVE | compatibility (V6, Oracle7, or the database version to which you are connected at precompile time) |
| DEFINE=<name> | none | a name for use by the Pro\*C preprocessor |
| DEF\_SQLCODE=YES|NO | NO | generate a macro to #define SQLCODE |
| ERRORS=YES | NO | YES | where to direct error messages (NO means only to listing file, and not to terminal) |
| FIPS=NONE | SQL89 | SQL2 | none | whether to flag ANSI/ISO non-compliance |
| HOLD\_CURSOR=YES |NO | NO | how cursor cache handles SQL statement |
| INAME=<filename> | none | name of the input file |
| INCLUDE=<pathname> | none | directory path for EXEC SQL INCLUDE or #include statements |
| LINES=YES | NO | NO | whether #line directives are generated |
| LNAME=<filename> | none | name of listing file |
| LTYPE=NONE|SHORT| LONG | LONG | type of listing file to be generated, if any |
| MAXLITERAL=10..1024 | 1024 | maximum length (bytes) of string literals in generated C code |
| MAXOPENCURSORS=5..255 | 10 | number of concurrent cached open cursors |
| MODE=ANSI|ISO|ORACLE | ORACLE | ANSI/ISO or Oracle behavior |
| NLS\_CHAR=(<var1>, ..., <varn>) | none | specify NLS character variables |
| NLS\_LOCAL=YES|NO | NO | control NLS character semantics |
| ONAME=<filename> | NONE | name of the output (code) file |
| ORACA=YES|NO | NO | whether to use the ORACA |
| PARSE=NONE | PARTIAL | FULL | FULL | whether Pro\*C parses (with a C parser) the .pc source. |
| RELEASE\_CURSOR=YES|NO | NO | control release of cursors from cursor cache |
| SELECT\_ERROR=YES|NO | YES | flagging of SELECT errors |
| SQLCHECK=SEMANTICS|SYNTAX | SYNTAX | kind of compile time SQL checking |
| SYS\_INCLUDE=<pathname> | none | directory where system header files, such as iostream.h, are found |
| UNSAFE\_NULL=YES|NO | NO | UNSAFE\_NULL=YES disables the ORA-01405 message |
| USERID=<username>/<password> | none | username/password[@dbname] connect string |
| VARCHAR=YES|NO | NO | allow the use of implicit VARCHAR structures |
|  |  |  |

**Table 7 - 1. Precompiler Options**

**Entering Options**

You can enter any precompiler option on the command line; many can also be entered inline in the precompiler program source file, using the EXEC ORACLE OPTION statement.

**On the Command Line**

You enter precompiler options on the command line using the following syntax:

... [OPTION\_NAME=value] [OPTION\_NAME=value] ...

Separate each option=value specification with one or more spaces. For example, you might enter the following:

... CODE=ANSI\_C MODE=ANSI

**Inline**

You enter options inline by coding EXEC ORACLE statements, using the following syntax:

EXEC ORACLE OPTION (OPTION\_NAME=value);

For example, you might code the following:

EXEC ORACLE OPTION (RELEASE\_CURSOR=yes);

An option entered inline overrides the same option entered on the command line, or specified in a configuration file.

**Uses for EXEC ORACLE**

The EXEC ORACLE feature is especially useful for changing option values during precompilation. For example, you might want to change HOLD\_CURSOR and RELEASE\_CURSOR on a statement-by-statement basis. Appendix [C](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/apc.htm) shows you how to optimize runtime performance using inline options.

Specifying options inline or in a configuration file is also helpful if your operating system limits the number of characters you can enter on the command line.

**Scope of EXEC ORACLE**

An EXEC ORACLE statement stays in effect until textually superseded by another EXEC ORACLE statement specifying the same option. In the following example, HOLD\_CURSOR=NO stays in effect until superseded by HOLD\_CURSOR=YES:

char emp\_name[20];

int emp\_number, dept\_number;

float salary;

EXEC SQL WHENEVER NOT FOUND DO break;

EXEC ORACLE OPTION (HOLD\_CURSOR=NO);

EXEC SQL DECLARE emp\_cursor CURSOR FOR

SELECT empno, deptno FROM emp;

EXEC SQL OPEN emp\_cursor;

printf(

"Employee Number Department\n--------------------------\n");

for (;;)

{

EXEC SQL FETCH emp\_cursor INTO :emp\_number, :dept\_number;

printf("%d\t%d\n", emp\_number, dept\_number);

}

EXEC SQL WHENEVER NOT FOUND CONTINUE;

for (;;)

{

printf("Employee number: ");

scanf("%d", &emp\_number);

if (emp\_number == 0)

break;

EXEC ORACLE OPTION (HOLD\_CURSOR=YES);

EXEC SQL SELECT ename, sal

INTO :emp\_name, :salary

FROM emp WHERE empno = :emp\_number;

printf("Salary for %s is %6.2f.\n", emp\_name, salary);

}

**Using the Precompiler Options**

This section is organized for easy reference. It lists the precompiler options alphabetically, and for each option gives its purpose, syntax, and default value. Usage notes that help you understand how the option works are also provided.

**AUTO\_CONNECT**

**Purpose**

Allows automatic connection to the OPS$ account.

**Syntax**

AUTO\_CONNECT={YES | NO}

**Default**

NO

**Usage Notes**

Can be entered only on the command line or in a configuration file.

If AUTO\_CONNECT=YES, and the application is not already connected to a database when it processes the first executable SQL statement, it attempts to connect using the userid

OPS$<username>

where *username* is your current operating system user or task name and OPS$username is a valid Oracle userid.

When AUTO\_CONNECT=NO, you must use the CONNECT statement in your program to connect to Oracle.

**CODE**

**Purpose**

Specifies the format of C function prototypes generated by the Pro\*C precompiler. (A *function prototype* declares a function and the datatypes of its arguments.) The precompiler generates function prototypes for SQL library routines, so that your C compiler can resolve external references. The CODE option lets you control the prototyping.

**Syntax**

CODE={ANSI\_C|KR\_C | CPP}

**Default**

KR\_C

**Usage Notes**

Can be entered inline or on the command line.

ANSI C standard X3.159-1989 provides for function prototyping. When CODE=ANSI\_C, Pro\*C/C++ generates full function prototypes, which conform to the ANSI C standard. An example follows:

extern void sqlora(long \*, void \*);

The precompiler can also generate other ANSI-approved constructs such as the **const** type qualifier.

When CODE=KR\_C (the default), the precompiler comments out the argument lists of generated function prototypes, as shown here:

extern void sqlora(/\*\_ long \*, void \* \_\*/);

So, specify CODE=KR\_C if your C compiler is not compliant with the X3.159 standard.

When CODE=CPP, the precompiler generates C++ compatible code.

**COMP\_CHARSET**

**Purpose**

Indicates to the Pro\*C/C++ Precompiler whether multi-byte character sets are (or are not) supported by the compiler to be used. It is intended for use by developers working in a multi-byte client-side environment (for example, when NLS\_LANG is set to a multi-byte character set).

**Syntax**

COMP\_CHARSET={MULTI\_BYTE|SINGLE\_BYTE}

**Default**

MULTI\_BYTE

**Usage Notes**

Can be entered only on the command line.

With COMP\_CHARSET=MULTI\_BYTE (default), Pro\*C/C++ generates C code that is to be compiled by a compiler that supports multi-byte NLS character sets.

With COMP\_CHARSET=SINGLE\_BYTE, Pro\*C/C++ generates C code for single-byte compilers that addresses a complication that *may* arise from the ASCII equivalent of a backslash (\) character in the second byte of a double-byte character in a multi-byte string. In this case, the backslash (\) character is "escaped" with another backslash character preceding it.

**Note**: The need for this feature is common when developing in a Shift-JIS environment with older C compilers.

This option has no effect when NLS\_LANG is set to a single-byte character set.

**CONFIG**

**Purpose**

Specifies the name of a user configuration file.

**Syntax**

CONFIG=<filename>

**Default**

None

**Usage Notes**

Can be entered only on the command line.

This option is the only way you can inform Pro\*C/C++ of the name and location of user configuration files.

**CPP\_SUFFIX**

**Purpose**

The CPP\_SUFFIX option allows you to specify the filename extension that the precompiler appends to the C++ output file generated when the CODE=CPP option is specified.

**Syntax**

CPP\_SUFFIX=<filename extension>

**Default**

System-specific.

**Usage Notes**

Most C compilers expect a default extension of ``.c'' for their input files. Different C++ compilers, however, can expect different filename extensions. The CPP\_SUFFIX option allows you to specify the filename extension that the precompiler generates. The value of this option is a string, without the quotes or the period. For example, CPP\_SUFFIX=cc, or CPP\_SUFFIX=C.

**DBMS**

**Purpose**

Specifies whether Oracle follows the semantic and syntactic rules of Oracle Version 6, Oracle7, or the native version of Oracle (that is, the version to which the application is connected).

**Syntax**

DBMS={NATIVE|V6|V7|V6\_CHAR}

**Default**

NATIVE

**Usage Notes**

Can be entered only on the command line, or in a configuration file.

The DBMS option lets you control the version-specific behavior of Oracle. When DBMS=NATIVE (the default), Oracle follows the semantic and syntactic rules of the database version to which the application is connected.

When DBMS=V6, V6\_CHAR, or DBMS=V7, Oracle follows the respective rules for Oracle Version 6 or Oracle7. A summary of the differences between DBMS=V6, DBMS=V6\_CHAR, and DBMS=V7 follows:

* When DBMS=V6 or V6\_CHAR, Oracle treats string literals like variable-length character values. However, when DBMS=V7, Oracle treats string literals like fixed-length character values, and CHAR semantics change slightly to comply with the current SQL standard.
* When DBMS=V6, Oracle treats local CHAR variables in a PL/SQL block like variable-length character values. When DBMS=V6\_CHAR, however, Oracle treats the CHAR variables like SQL standard, fixed-length character values.
* When DBMS=V6 or V6\_CHAR, Oracle treats the return value of the SQL function USER like a variable-length character value. However, when DBMS=V7, Oracle treats the return value of USER like a SQL standard, fixed-length character value.
* When DBMS=V6 or V6\_CHAR, the default Oracle external datatype for variables that have the C type **char**or **char[n]** is CHAR. When DBMS=V7, the default external datatype is CHARZ for **char[n]** and VARCHAR2 for **char**.
* When DBMS=V6 (but not V6\_CHAR), if you process a multirow query that calls a SQL group function such as AVG or COUNT, the function is called at OPEN time. When DBMS=V7 or V6\_CHAR, however, the function is called at FETCH time. At OPEN time or FETCH time, if the function call fails, Oracle issues an error message immediately. Thus, the DBMS value affects error reporting slightly.
* When DBMS=V6, no error is returned if a SELECT or FETCH statement selects a null, and there is no indicator variable associated with the output host variable. When DBMS=V7 or V6\_CHAR, SELECTing or FETCHing a null column or expression into a host variable that has no associated indicator variable causes an error (SQLSTATE is "22002"; SQLCODE is -01405).
* When DBMS={V6|V6\_CHAR}, a DESCRIBE operation of a fixed-length string (in Dynamic SQL Method 4) returns datatype code 1. When DBMS=V7, the DESCRIBE operation returns datatype code 96.
* When DBMS=V6, PCTINCREASE is allowed for rollback segments. When DBMS=V7 or V6\_CHAR, PCTINCREASE is not allowed for rollback segments.
* When DBMS=V6, illegal MAXEXTENTS storage parameters are allowed. They are not allowed when DBMS=V7 or V6\_CHAR.
* When DBMS=V6, constraints (except NOT NULL) are not enabled. When DBMS=V7 or V6\_CHAR, all Oracle7 constraints are enabled.

If you precompile using the DBMS=V6 option, and connect to an Oracle7 database, then a Data Definition Language statement such as

CREATE TABLE T1 (COL1 CHAR(10))

creates the table using the VARCHAR2 (variable-length) datatype, just as if the CREATE TABLE statement had been

CREATE TABLE T1 (COL1 VARCHAR2(10))

|  |  |  |  |
| --- | --- | --- | --- |
| **Situation** | **DBMS=V7|V6\_CHAR MODE=ANSI** | **DBMS=V7|V6\_CHAR MODE=ORACLE** | **DBMS=V6 MODE=ORACLE** |
| "no data found" warning code | +100 | +1403 | +1403 |
| fetch nulls without using indicator variables | error -1405 | error -1405 | no error |
| fetch truncated values without using indicator variables | no error but SQLWARN(2) is set | no error but SQLWARN(2) is set | error -1406 and SQLWARN(2) is set |
| cursors closed by COMMIT or ROLLBACK | all explicit | CURRENT OF only | CURRENT OF only () |
| open an already OPENed cursor | error -2117 | no error | no error |
| close an already CLOSEd cursor | error -2114 | no error | no error |
| SQL group function ignores nulls | no warning | no warning | SQLWARN(3) is set |
| when SQL group function in multirow query is called | FETCH time | FETCH time | OPEN time |
| declare SQLCA structure | optional | required | required () |
| declare SQLCODE or SQLSTATE status variable | required | optional but Oracle ignores | optional but Oracle ignores () |
| integrity constraints | enabled | enabled | disabled |
| PCTINCREASE for rollback segments | not allowed | not allowed | allowed |
| MAXEXTENTS storage parameters | not allowed | not allowed | allowed |
|  |  |  |  |

**Table 7 - 2. How DBMS and MODE Interact**

**DEF\_SQLCODE**

**Purpose**

Controls whether the Pro\*C precompiler generates **#define**'s for SQLCODE.

**Syntax**

DEF\_SQLCODE={NO|YES}

**Default**

NO

**Usage Notes**

Can be used only on the command line or in a configuration file.

When DEF\_SQLCODE=YES, the precompiler defines SQLCODE in the generated source code as follows:

#define SQLCODE sqlca.sqlcode

You can then use SQLCODE to check the results of executable SQL statement. The DEF\_SQLCODE option is supplied for compliance with standards that require the use of SQLCODE.

In addition, you must also include the SQLCA using one of the following entries in your source code:

#include <sqlca.h>

or

EXEC SQL INCLUDE SQLCA;

If the SQLCA is not included, using this option causes a precompile time error.

**DEFINE**

**Purpose**

Defines a name that can be used in **#ifdef**and **#ifndef** Pro\*C preprocessor directives. The defined name can also be used by the EXEC ORACLE IFDEF and EXEC ORACLE IFNDEF statements.

**Syntax**

DEFINE=*name*

**Default**

None

**Usage Notes**

Can be entered on the command line or inline. You can only use DEFINE to define a name--you cannot define macros with it. For example, the following use of define is not valid:

proc my\_prog DEFINE=LEN=20

Using DEFINE in the correct way, you could do

proc my\_prog DEFINE=XYZZY

And then in *my\_prog.pc*, code

#ifdef XYZZY

...

#else

...

#endif

Or you could just as well code

EXEC ORACLE IFDEF XYZZY;

...

EXEC ORACLE ELSE;

...

EXEC ORACLE ENDIF;

The following example is *invalid*:

#define XYZZY

...

EXEC ORACLE IFDEF XYZZY

...

EXEC ORACLE ENDIF;

EXEC ORACLE conditional statements are *valid* only if the macro is defined using EXEC ORACLE DEFINE or the DEFINE option.

If you define a name using DEFINE=, and then conditionally include (or exclude) a code section using the Pro\*C preprocessor **#ifdef** (or **#ifndef**) directives, you must also make sure that the name is defined when you run the C compiler. For example, for UNIX *cc*, you must use the -D option to define the name for the C compiler.

**ERRORS**

**Purpose**

Specifies whether error messages are sent to the terminal as well as the listing file (YES), or just to the listing file (NO).

**Syntax**

ERRORS={YES|NO}

**Default**

YES

**Usage Notes**

Can be entered only on the command line, or in a configuration file.

**FIPS**

**Purpose**

Specifies whether extensions to ANSI SQL are flagged (by the FIPS Flagger). An extension is any SQL element that violates ANSI format or syntax rules, except privilege enforcement rules.

**Syntax**

FIPS={NONE|SQL89|SQL2|YES|NO}

**Default**

None

**Usage Notes**

Can be entered inline or on the command line.

When FIPS=YES, the FIPS Flagger is enabled, and warning (not error) messages are issued if you use an Oracle extension to ANSI SQL, or use an ANSI SQL feature in a nonconforming manner. Extensions to ANSI SQL that are flagged at precompile time include the following:

* array interface including the FOR clause
* SQLCA, ORACA, and SQLDA data structures
* dynamic SQL including the DESCRIBE statement
* embedded PL/SQL blocks
* automatic datatype conversion
* DATE, NUMBER, RAW, LONGRAW, VARRAW, ROWID, VARCHAR2, and VARCHAR datatypes
* pointer host variables
* Oracle OPTION statement for specifying runtime options
* IAF statements in user exits
* CONNECT statement
* TYPE and VAR datatype equivalencing statements
* AT <db\_name> clause
* DECLARE...DATABASE, ...STATEMENT, and ...TABLE statements
* SQLWARNING condition in WHENEVER statement
* DO *function\_name()* and DO **break** actions in WHENEVER statement
* COMMENT and FORCE TRANSACTION clauses in COMMIT statement
* FORCE TRANSACTION and TO SAVEPOINT clauses in ROLLBACK statement
* RELEASE parameter in COMMIT and ROLLBACK statements
* optional colon-prefixing of WHENEVER...GOTO labels, and of host variables in the INTO clause

A sample of Pro\*C/C++ output when FIPS=YES is shown below.

Pro\*C/C++: Release 2.1.1.0.0 - Beta on Thu Sep 22 14:28:43 1994

Copyright (c) Oracle Corporation 1979, 1994. All rights reserved.

System default option values taken from: /private2/dve/k72tt/proc/pcscfg.h

Oracle FIPS Flagging Report Version 1.0 - Development

This report lists extensions to ANSI SQL document X3.168-1989

The following extensions were detected:

Violation Line Number Description

--------- ----------- -------------------------------------

00707 106 keyword WORK required after ROLLBACK

00709 106 use of RELEASE clause

00710 65 use of dynamic SQL

00717 28 56 use of DO within WHENEVER clause

00724 22 53 invalid datatype

The following non-standard usages were detected:

No extensions were detected.

The following extensions which will become standard in a future

release of the SQL standard were detected:

Violation Revision Line Number Description

--------- ----------- ----------- --------------------------

00704 SQL 2 Draft 30 use of the CONNECT statement

The following deprecated features were detected:

No extensions were detected.

Found 6 violations of ANSI SQL standard X3.168-1989.

ANSI SQL document comparison cross-reference by line number:

Line Numbers Document Referenced

------------ -------------------

1 - end X3.168-1989

The Oracle FIPS Flagger was active for all lines in the source.

**HOLD\_CURSOR**

**Purpose**

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

**Syntax**

HOLD\_CURSOR={YES|NO}

**Default**

NO

**Usage Notes**

Can be entered inline or on the command line.

You can use HOLD\_CURSOR to improve the performance of your program. For more information, see Appendix [C](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/apc.htm).

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. HOLD\_CURSOR controls what happens to the link between the cursor and cursor cache.

When HOLD\_CURSOR=NO, after Oracle executes the SQL statement and the cursor is closed, the precompiler marks the link as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks.

When HOLD\_CURSOR=YES and RELEASE\_CURSOR=NO, the link is maintained; the precompiler does not reuse it. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set HOLD\_CURSOR before executing the SQL statement. For inline use with explicit cursors, set HOLD\_CURSOR before CLOSEing the cursor.

Note that RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES and that HOLD\_CURSOR=NO overrides RELEASE\_CURSOR=NO. For information showing how these two options interact, see [Table C - 1](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/apc.htm#CurInter).

**INAME**

**Purpose**

Specifies the name of the input file.

**Syntax**

INAME=<path and filename>

**Default**

None

**Usage Notes**

Can be entered only on the command line.

You can omit the filename extension if is*.pc*. If the input filename is the first option on the command line, you can omit the INAME= part of the option. For example:

proc sample1 MODE=ansi

to precompile the file *sample1.pc*, using ANSI mode. This command is the same as

proc INAME=sample1 MODE=ansi

**INCLUDE**

**Purpose**

Specifies a directory path for files included using the **#include** or EXEC SQL INCLUDE directives.

**Syntax**

INCLUDE=*pathname* or INCLUDE=(*path\_1*,*path\_2*,...,*path\_n*)

**Default**

Current directory and paths built into Pro\*C

**Usage Notes**

Can be entered inline or on the command line.

You use INCLUDE to specify a directory path for included files. The precompiler searches directories in the following order:

1. the directory specified in a SYS\_INCLUDE precompiler option

2. the current directory

3. the built-in directory for standard header files

4. the directory specified by the INCLUDE option

Because of step [3](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#4BF8jc5tssm), you normally do not need to specify a directory path for standard header files such as *sqlca.h* and *sqlda.h*. (On UNIX systems, the precompiler searches for these files in *$ORACLE\_HOME/sqllib/public*.)

**Note**: If you specify a filename without an extension for inclusion, Pro\*C assumes an extension of *.h*. So, files to be included should have an extension, even if it is not .*h*.

You must still use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. You can specify more than one path on the command line, as follows:

... INCLUDE=<path\_1> INCLUDE=<path\_2> ...

The precompiler searches first in the current directory, then in the directory for standard header files, and finally in the directory named by *path1*, then in the directory named by *path2*.

**Warning**: The precompiler looks for a file in the current directory first--even if you specify a directory path. So, if the file you want to include resides in another directory, make sure no file with the same name resides in the current directory.

The syntax for specifying a directory path using the INCLUDE option is system specific. Follow the conventions used for your operating system.

**LINES**

**Purpose**

Specifies whether the Pro\*C precompiler adds **#line** preprocessor directives to its output file.

**Syntax**

LINES={YES|NO}

**Default**

NO

**Usage Notes**

Can be entered only on the command line.

The LINES option helps with debugging. When LINES=YES, the Pro\*C precompiler adds **#line** preprocessor directives to its output file.

Normally, your C compiler increments its line count after each input line is processed. The **#line** directives force the compiler to reset its input line counter so that lines of precompiler-generated code are not counted. Moreover, when the name of the input file changes, the next **#line** directive specifies the new filename.

The C compiler uses the line numbers and filenames to show the location of errors. Thus, error messages issued by the C compiler always refer to your original source files, not the modified source file.

When LINES=NO (the default), the precompiler adds no **#line** directives to its output file.

**Note**: On page [3 - 3](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch03a.htm#linedir), it is stated that the Pro\*C precompiler does not support the **#line** directive. This means that you cannot directly code **#line** directives in the precompiler source. But you can still use the LINES= option to have the precompiler insert **#line** directives for you.

**LNAME**

**Purpose**

Specifies the name of the listing file.

**Syntax**

LNAME=<filename>

**Default**

None

**Usage Notes**

Can be entered only on the command line.

The default filename extension for the listing file is *.lis*.

**LTYPE**

**Purpose**

Specifies the type of listing file generated.

**Syntax**

LTYPE={NONE|SHORT|LONG}

**Default**

LONG

**Usage Notes**

Can be entered on the command line or in a configuration file.

When a listing file is generated, the LONG format is the default. With LTYPE=LONG specified, all of the source code is listed as it is parsed and messages listed as they are generated. In addition, the Pro\*C/C++ currently in effect are listed.

With LTYPE=SHORT specified, only the generated messages are listed--no source code--with line references to the source file to help you locate the code that generated the message condition.

With LTYPE=NONE specified, no list file is produced *unless* the LNAME option explicitly specifies a name for a list file. Under the latter condition, the list file *is* generated with LTYPE=LONG assumed.

**MAXLITERAL**

**Purpose**

Specifies the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded.

**Syntax**

MAXLITERAL=integer, range is 10 to 1024

**Default**

1024

**Usage Notes**

Cannot be entered inline.

The maximum value of MAXLITERAL is compiler dependent. For example, some C compilers cannot handle string literals longer than 512 characters, so you would specify MAXLITERAL=512.

Strings that exceed the length specified by MAXLITERAL are divided during precompilation, then recombined (concatenated) at run time.

**MAXOPENCURSORS**

**Purpose**

Specifies the number of concurrently open cursors that the precompiler tries to keep cached.

**Syntax**

MAXOPENCURSORS=*integer*

**Default**

10

**Usage Notes**

Can be entered inline or on the command line.

You can use MAXOPENCURSORS to improve the performance of your program. For more information, see Appendix [C](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/apc.htm).

When precompiling separately, use MAXOPENCURSORS as described in "Guidelines for Precompiling Separately" [[*]](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#separate).

MAXOPENCURSORS specifies the *initial* size of the SQLLIB cursor cache. If a new cursor is needed, and there are no free cache entries, Oracle tries to reuse an entry. Its success depends on the values of HOLD\_CURSOR and RELEASE\_CURSOR, and, for explicit cursors, on the status of the cursor itself. Oracle allocates an additional cache entry if it cannot find one to reuse.

If necessary, Oracle keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN\_CURSORS. MAXOPENCURSORS must be lower than OPEN\_CURSORS by at least 6 to avoid a "maximum open cursors exceeded" Oracle error.

As your program's need for concurrently open cursors grows, you might want to respecify MAXOPENCURSORS to match the need. A value of 45 to 50 is not uncommon, but remember that each cursor requires another private SQL area in the user process memory space. The default value of 10 is adequate for most programs.

**MODE**

**Purpose**

Specifies whether your program observes Oracle practices or complies with the current ANSI/ISO SQL standards.

**Syntax**

MODE={ANSI|ISO|ORACLE}

**Default**

ORACLE

**Usage Notes**

Can be entered only on the command line or in a configuration file.

ISO is a synonym for ANSI.

When MODE=ORACLE (the default), your embedded SQL program observes Oracle practices. When MODE=ANSI, your program complies *fully* with the ANSI SQL standard, and the following changes go into effect:

* Issuing a COMMIT or ROLLBACK closes all explicit cursors.
* You cannot OPEN an already open cursor or CLOSE an already closed cursor. (When MODE=ORACLE, you can reOPEN an open cursor to avoid reparsing.)
* You must declare a either a **long** variable named *SQLCODE* or a **char**SQLSTATE[6]variable (uppercase is required for both variables) that is in scope of every EXEC SQL statement. The same *SQLCODE* or *SQLSTATE*variable need not be used in each case; that is, the variable need not be global.
* Declaring the SQLCA is optional. You need not include the SQLCA.
* The ``no data found" Oracle warning code returned to SQLCODE becomes +100 instead of +1403. The message text does not change.

**NLS\_CHAR**

**Purpose**

Specifies which C host character variables are treated by the precompiler as National Language Support (NLS) multi-byte character variables.

**Syntax**

NLS\_CHAR=*varname* or NLS\_CHAR=(*var\_1*,*var\_2*,...,*var\_n*)

**Default**

None.

**Usage Notes**

Can be entered only on the command line, or in a configuration file.

This option allows you to specify at precompile time a list of the names of one or more host variables that the precompiler must treat as National Language character variables. You can specify only C **char** variables or Pro\*C/C++ VARCHARs using this option.

If a you specify in the option list a variable that is not declared in your program, then the precompiler generates no error.

**NLS\_LOCAL**

**Purpose**

Determines whether NLS character conversions are performed by the precompiler runtime library, or by the Oracle Server.

**Syntax**

NLS\_LOCAL={NO|YES}

**Default**

NO

**Usage Notes**

Can be entered only on the command line, or in a configuration file.

When NLS\_LOCAL=YES, the runtime library (SQLLIB) performs blank-padding and blank-stripping for host variables that are National Language Support (NLS) multi-byte types.

When NLS\_LOCAL=NO, the Oracle Server performs these actions.

When you use the NLS\_CHAR option to specify multi-byte character host variables, you must specify NLS\_LOCAL=YES.

**ONAME**

**Purpose**

Specifies the name of the output file. The output file is the C code file that the precompiler generates.

**Syntax**

ONAME=<filename>

**Default**

INAME with an extension determined by CPP\_SUFFIX.

**Usage Notes**

Can be entered only on the command line. Use this option to specify the name of the output file, where the name differs from that of the input (.*pc*) file. For example, if you issue the command

proc iname=my\_test

the default output filename is *my\_test.c*. If you want the output filename to be *my\_test\_1.c*, issue the command

proc iname=my\_test oname=my\_test\_1.c

Note that you should add the*.c*extension to files specified using ONAME.

The default extension with the ONAME option is platform-specific, but you can override it using the CODE and CPP\_SUFFIX options. When CODE=KR\_C or ANSI\_C, the extension is *c*. When CODE=CPP, you can use the CPP\_SUFFIX option to override the platform-specific default.

**Attention**: Oracle recommends that you not let the output filename default, but rather name it explicitly using ONAME.

**ORACA**

**Purpose**

Specifies whether a program can use the Oracle Communications Area (ORACA).

**Syntax**

ORACA={YES|NO}

**Default**

NO

**Usage Notes**

Can be entered inline or on the command line.

When ORACA=YES, you must place either the EXEC SQL INCLUDE ORACA or **#include***oraca.h* statement in your program.

**PARSE**

**Purpose**

Specifies the way that the Pro\*C precompiler parses the source file.

**Syntax**

PARSE={FULL|PARTIAL|NONE}

**Default**

FULL

**Usage Notes**

To generate C++ compatible code, the PARSE option must be either NONE or PARTIAL. If PARSE=FULL, the C parser runs, and it does not understand C++ constructs in your code, such as classes.

See page [6 - 5](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch06.htm#c++-parse) for more information on the PARSE option.

With PARSE=FULL or PARSE=PARTIAL Pro\*C/C++ fully supports C preprocessor directives, such as **#define**, **#ifdef**, and so on. However, with PARSE=NONE conditional preprocessing is supported by EXEC ORACLE statements as described in "Conditional Preprocessing" [[*]](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#condition).

**RELEASE\_CURSOR**

**Purpose**

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

**Syntax**

RELEASE\_CURSOR={YES|NO}

**Default**

NO

**Usage Notes**

Can be entered inline or on the command line.

You can use RELEASE\_CURSOR to improve the performance of your program. For more information, see Appendix [C](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/apc.htm).

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. RELEASE\_CURSOR controls what happens to the link between the cursor cache and private SQL area.

When RELEASE\_CURSOR=YES, after Oracle executes the SQL statement and the cursor is closed, the precompiler immediately removes the link. This frees memory allocated to the private SQL area and releases parse locks. To make sure that associated resources are freed when you CLOSE a cursor, you must specify RELEASE\_CURSOR=YES.

When RELEASE\_CURSOR=NO and HOLD\_CURSOR=YES, the link is maintained. The precompiler does not reuse the link unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set RELEASE\_CURSOR before executing the SQL statement. For inline use with explicit cursors, set RELEASE\_CURSOR before CLOSEing the cursor.

Note that RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES and that HOLD\_CURSOR=NO overrides RELEASE\_CURSOR=NO. For a table showing how these two options interact, see Appendix [C](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/apc.htm).

**SELECT\_ERROR**

**Purpose**

Specifies whether your program generates an error when a SELECT statement returns more than one row, or more rows than a host array can accommodate.

**Syntax**

SELECT\_ERROR={YES|NO}

**Default**

YES

**Usage Notes**

Can be entered inline or on the command line.

When SELECT\_ERROR=YES, an error is generated when a single-row SELECT returns too many rows, or when an array SELECT returns more rows than the host array can accommodate. The result of the SELECT is indeterminate.

When SELECT\_ERROR=NO, no error is generated when a single-row SELECT returns too many rows, or when an array SELECT returns more rows than the host array can accommodate.

Whether you specify YES or NO, a random row is selected from the table. The only way to ensure a specific ordering of rows is to use the ORDER BY clause in your SELECT statement. When SELECT\_ERROR=NO and you use ORDER BY, Oracle returns the first row, or the first *n* rows when you are SELECTing into an array. When SELECT\_ERROR=YES, whether or not you use ORDER BY, an error is generated when too many rows are returned.

**SQLCHECK**

**Purpose**

Specifies the type and extent of syntactic and semantic checking.

**Syntax**

SQLCHECK={SEMANTICS|FULL|SYNTAX|LIMITED|NONE}

**Default**

SYNTAX

**Usage Notes**

Can be entered inline or the command line.

The Pro\*C precompiler can help you debug a program by checking the syntax and semantics of embedded SQL statements and PL/SQL blocks. You control the level of checking by entering the SQLCHECK option inline and/or on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line. For example, if you specify SQLCHECK=SYNTAX on the command line, you cannot specify SQLCHECK=SEMANTICS inline.

**SQLCHECK=SEMANTICS|FULL** The precompiler checks the syntax and semantics of

* data manipulation statements such as INSERT and UPDATE
* PL/SQL blocks
* host variable datatypes

However, only syntactic checking is done on data manipulation statements or PL/SQL blocks that use the AT *db\_name* clause. No syntax or semantics checking is performed on DDL statements, such as CREATE and ALTER.

Any errors found are reported at precompile time.

The precompiler gets information needed for a semantic check by using embedded DECLARE TABLE statements, or if you specify the USERID option on the command line, by connecting to Oracle and accessing the data dictionary. You need not connect to Oracle if every table referenced in a data manipulation statement or PL/SQL block is defined in a DECLARE TABLE statement.

If you connect to Oracle, but some needed information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. A DECLARE TABLE definition overrides a data dictionary definition if they conflict.

If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK=SEMANTICS and the USERID option as well.

**SQLCHECK=SYNTAX|LIMITED|NONE** The precompiler checks the syntax of

* declarative SQL statements (such as EXEC SQL WHENEVER...)
* Data Manipulation Language statements
* host variables and host variable datatypes

and any errors found are reported at precompile time.

But no semantic checking is done. DECLARE TABLE statements are ignored, and PL/SQL blocks are not allowed.

Specifying the SYNTAX value generates a useable output (code) file, however semantic errors can still occur at runtime.

**SYS\_INCLUDE**

**Purpose**

Specifies the location of system header files.

**Syntax**

SYS\_INCLUDE=<pathname>

**Default**

System-specific.

**Usage Notes**

Pro\*C searches for standard system header files, such as *stdio.h*, in standard locations that are platform specific. For example, on almost all UNIX systems, the file *stdio.h* has the full pathname */usr/include/stdio.h.*

But C++ compilers can have system header files, such as *stdio.h*, that are not in the standard system locations. You can use the SYS\_INCLUDE command line option to specify a list of directory paths that Pro\*C searches to look for system header files. For example

SYS\_INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)

The search path that you specify using SYS\_INCLUDE overrides the default header location

If PARSE=NONE, the value specified in SYS\_INCLUDE is irrelevant, since there is no need for Pro\*C to include system header files. (You must, of course, still include Pro\*C-specific headers, such *sqlca.h*.)

The precompiler searches directories in the following order:

1. the directory specified in the SYS\_INCLUDE precompiler option

2. the current directory

3. the built-in directory for standard header files

4. the directory specified by the INCLUDE option

Because of step [3](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch07.htm#4BF8jc5tssm), you normally do not need to specify a directory path for standard header files such as *sqlca.h* and *sqlda.h*. (On UNIX systems, the precompiler searches the *$ORACLE\_HOME/sqllib/public* directory for these files.)

**UNSAFE\_NULL**

**Purpose**

Specifying UNSAFE\_NULL=YES prevents generation of ORA-01405 messages when fetching NULLs without using indicator variables.

**Syntax**

UNSAFE\_NULL={YES|NO}

**Default**

NO

**Usage Notes**

Cannot be entered inline.

The UNSAFE\_NULL=YES is allowed only when MODE=ORACLE and DBMS=V7 or V6\_CHAR.

The UNSAFE\_NULL option has no effect on host variables in an embedded PL/SQL block. You *must* use indicator variables to avoid ORA-01405 errors.

**USERID**

**Purpose**

Specifies an Oracle username and password.

**Syntax**

USERID=username/password

**Default**

None

**Usage Notes**

Can be entered only on the command line.

Do not specify this option when using the automatic connect feature, which accepts your Oracle username prefixed with OPS$. The actual value of the "OPS$" string is set as a parameter in the INIT.ORA file.

When SQLCHECK=SEMANTICS, if you want the precompiler to get needed information by connecting to Oracle and accessing the data dictionary, you must also specify USERID.

**THREADS**

**Purpose**

When THREADS=YES, the precompiler searches for context declarations.

**Syntax**

THREADS={YES|NO}

**Default**

NO

**Usage Notes**

Cannot be entered inline.

This precompiler option is required for any program that requires multi-threaded support.

With THREADS=YES, the precompiler generates an error if no EXEC SQL USE directive is encountered before the first context is visible and an executable SQL statement is found. For more information, see "Developing Multi-threaded Applications" [[*]](https://docs.oracle.com/cd/A57673_01/DOC/api/doc/PC_22/ch03a.htm#mt_apps).

**VARCHAR**

**Purpose**

Instructs the Pro\*C precompiler to interpret some structs as VARCHAR host variables.

**Syntax**

VARCHAR={NO|YES}

**Default**

NO

**Usage Notes**

Can be entered only on the command line.

When VARCHAR=YES, a C struct that you code as

struct {

short <len>;

char <arr>[n];

} name;

is interpreted by the precompiler as a **VARCHAR[n]** host variable.

**Obsolete Options**

The following precompiler options, which are simply parsed and ignored, are not supported in Pro\*C/C++ release 2.2:

* ASACC
* IRECLEN
* LRECLEN
* ORECLEN
* PAGELEN
* TEST
* XREF

With Oracle7 and Pro\*C Release 1.5 or later, private SQL areas are automatically resized, host variables are rebound only when necessary, and reentrant code is generated automatically for systems that require it. These advances make the AREASIZE, REBIND, and REENTRANT options obsolete. You no longer have to worry about using these options correctly. In fact, if you specify AREASIZE, REBIND, or REENTRANT, you get the following informational message:

PCC-I-02355: Invalid or obsolete option, ignored

**AREASIZE**

With some earlier releases of Pro\*C, the AREASIZE option specified the size of the initial private SQL area opened for Oracle cursors. You could respecify AREASIZE for each cursor or set of cursors used by your program.

**REBIND**

The REBIND option specified how often host variables in SQL statements were bound. You could respecify REBIND for each SQL statement or set of SQL statements in your program.

**REENTRANT**

The REENTRANT option specified whether reentrant code was generated. (A *reentrant* program or subroutine can be reentered before it has finished executing. Thus, it can be used simultaneously by two or more processes.) On some systems, you had to specify REENTRANT=YES.

**Conditional Precompilations**

Conditional precompilation includes (or excludes) sections of code in your C program based on certain conditions. For example, you might want to include one section of code when precompiling under UNIX and another when precompiling under VMS. Conditional precompiling lets you write programs that can run in different environments.

Conditional sections of code are marked by statements that define the environment and actions to take. You can code C statements as well as EXEC SQL statements in these sections. The following statements let you exercise conditional control over precompilation:

EXEC ORACLE DEFINE symbol; -- define a symbol

EXEC ORACLE IFDEF symbol; -- if symbol is defined

EXEC ORACLE IFNDEF symbol; -- if symbol is not defined

EXEC ORACLE ELSE; -- otherwise

EXEC ORACLE ENDIF; -- end this control block

All EXEC ORACLE statements must be terminated with a semi-colon.

**Defining Symbols**

You can define a symbol in two ways. Either include the statement

EXEC ORACLE DEFINE symbol;

in your host program or define the symbol on the command line using the syntax

... INAME=filename ... DEFINE=symbol

where *symbol* is not case-sensitive.

**Warning**: The **#define** preprocesssor directive is not the same as the EXEC ORACLE DEFINE command

Some port-specific symbols are predefined for you when the Pro\*C Precompiler is installed on your system. For example, predefined operating system symbols include CMS, MVS, MS-DOS, UNIX, and VMS.

**An Example**

In the following example, the SELECT statement is precompiled only when the symbol *site2* is defined:

EXEC ORACLE IFDEF site2;

EXEC SQL SELECT DNAME

INTO :dept\_name

FROM DEPT

WHERE DEPTNO = :dept\_number;

EXEC ORACLE ENDIF;

Blocks of conditions can be nested as shown in the following example:

EXEC ORACLE IFDEF outer;

EXEC ORACLE IFDEF inner;

...

EXEC ORACLE ENDIF;

EXEC ORACLE ENDIF;

You can "comment out" C or embedded SQL code by placing it between IFDEF and ENDIF and *not* defining the symbol.

**Guidelines for Precompiling Separately**

The following guidelines will help you avoid some common problems.

**Referencing Cursors**

Cursor names are SQL identifiers, whose scope is the precompilation unit. Hence, cursor operations cannot span precompilation units (files). That is, you cannot DECLARE a cursor in one file, and OPEN or FETCH from it in another file. So, when doing a separate precompilation, make sure all definitions and references to a given cursor are in one file.

**Specifying MAXOPENCURSORS**

When you precompile the program module that CONNECTs to Oracle, specify a value for MAXOPENCURSORS that is high enough for any of the program modules. If you use MAXOPENCURSORS for another program module, one that does not do a CONNECT, then that value for MAXOPENCURSORS is ignored. Only the value in effect for the CONNECT is used at run time.

**Using a Single SQLCA**

If you want to use just one SQLCA, you must declare it as global in one of the program modules and as external in the other modules. Use the **extern**storage class, and the following define in your code:

#define SQLCA\_STORAGE\_CLASS extern

which tell the precompiler to look for the SQLCA in another program module. Unless you declare the SQLCA as external, each program module uses its own local SQLCA.

**Compiling and Linking**

To get an executable program, you must compile the output *.c* source files produced by the precompiler, then link the resulting object modules with modules needed from SQLLIB and system-specific Oracle libraries. If you are mixing precompiler code and OCI calls, be sure to also link in the OCI runtime library (*liboci.a* on UNIX systems).

The linker resolves symbolic references in the object modules. If these references conflict, the link fails. This can happen when you try to link third-party software into a precompiled program. Not all third-party software is compatible with Oracle. So, linking your program *shared* might cause an obscure problem. In some cases, linking *stand-alone* or *two-task* might solve the problem.

Compiling and linking are system dependent. On most platforms, example *makefiles* or batch files are supplied that you can use to precompile, compile, and link a Pro\*C application. See your system-specific Oracle documentation.